A symbolic approach to the state graph based analysis of high-level Markov reward models

Ein symbolischer Ansatz für die Zustandsgraph-basierte Analyse von hochsprachlichen Markov Reward Modellen

Der Technischen Fakultät
der Universität Erlangen-Nürnberg
zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von
Kai Matthias Lampka

Erlangen, März 2007

Als Dissertation genehmigt von
der Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung: 27.11.2006

Tag der Promotion: 19.3.2007

Dekan: Prof. Dr.-Ing. Alfred Leipertz

Berichterstatter: Prof. Dr.-Ing. Markus Siegle (Univ. der Bundeswehr München)
Prof. Dr.-Ing. Reinhard German (Univ. Erlangen-Nürnberg)
Prof. Ph.D. William H. Sanders (Univ. of Illinois, Urbana-Champaign)

Abstract

Markov reward models considered in this thesis are compactly described by means of Markovian extensions of well-known high-level model description formalisms. For numerically computing performance and dependability (= performability) measures of high-level system models, the latter must be transformed into low-level representations, where the concurrency contained in the high-level model description is made explicit. This transformation, where a high-level model is mapped onto a (stochastic) state/transition-system, generically denoted as state graph (SG), may therefore yield an exponential blow-up in the number of system states. This problem is known as the notorious *state space explosion problem*. Decision diagrams (DD) have shown to be very helpful when it comes to the representation of extremely large SGs, easing the restriction imposed on the size and complexity of models and thus systems to be analyzed. However, to efficiently apply contemporary symbolic techniques the high level models must possess either a specific compositional structure and/or the employed modeling formalism must be of a specific kind. This work lift these limitations, where the number of system states, the state probability of wich must be computed, is still the limiting factor of the analysis.

To represent SGs, this thesis extends "zero-suppressed" binary decision diagrams to the case of "zero-suppressed" multi-terminal binary decision diagrams (ZDDs). To deduce the pseudo-boolean function represented by a ZDD's graph correctly, the set of Boolean function variables must be known. Consequently, within a shared DD-environment as it is provided by well-known DD-packages, ZDD-nodes lose their uniqueness. To solve this problem, the concept of partially shared ZDDs (*p*ZDDs) is introduced, so that nodes are extended with sets of function variables. It is shown that *p*ZDDs are canonical representations of pseudo-boolean functions. For efficiently working with *p*ZDDs, this thesis also develops a wide range of (symbolic) algorithms. These algorithms are designed in such a way that they allow to implement *p*ZDDs within common, shared DD-environments.
If a model description formalism does not possess a symbolic semantic, symbolic representations of annotated state/transition-systems can only be deduced from its high-level model descriptions by explicit execution. To do so in a memory and run-time efficient manner, this work exploits local information of high-level model constructs only, yielding the *activity/reward-local* approach. This new semi-symbolic technique comprises the four follwing steps: (a) The activity-local scheme for generating symbolic representation of a high-level model's SG. Since the suggested procedure does not generate all system states explicitly, the use of a symbolic composition scheme is required. The newly developed composition scheme delivers the potential SG and its restriction to the set of reachable transitions is efficiently achieved by making use of symbolic reachability analysis, where this thesis introduces a new "quasi" depth-first-search based algorithm. (b) The reward-local scheme for obtaining symbolic representations of reward functions as defined on the high-level model. Analogously to the above procedure, one explicitly executes the reward functions for evaluating the reward values of states and transitions. But for reducing the number of explicit state visits, the procedure once again exploits local information only. (c) For the computation of state probabilities, this work introduces a ZDD-based variant of the hybrid solution method, developed in the context of other symbolic data structures. (d) Given symbolically represented reward functions and state probabilities, as the next step, one determines the user-defined performability measures of the high-level model, where for this purpose a new graph-traversing algorithm is introduced.

Since the *activity/reward*-local scheme depends on explicit but in most cases partial execution it is not limited to a certain description technique. Based on a new symbolic composition scheme and contrary to other symbolic approaches, it is still applicable, if the high-level models are neither compositionally constructed nor possess a decomposable structure of a certain kind. Thus this thesis not only introduces a new type of decision diagram and algorithms for efficiently working with it, but also develops a universal symbolic approach for the SG based analysis of high-level Markov reward models with very large SGs.

Table of Contents

List of Figures IV

List of Algorithms V

List of Tables VI

1 Introduction 1
- 1.1 Motivation 1
- 1.2 State space explosion problem and related approaches 2
- 1.3 State-of-the symbolic techniques 4
- 1.4 Contributions of this thesis 7
- 1.5 Organization of the thesis 7

2 Background Material 9
- 2.1 Organization of the chapter 9
- 2.2 Markov Theory 9
  - 2.2.1 Continuous-time Markov reward model (MRM) 9
  - 2.2.2 Numerical solution of MRM 11
  - 2.2.3 Reduction techniques 16
  - 2.2.4 State/Transition systems 19
- 2.3 High-level Markov reward models 20
  - 2.3.1 High-level model description techniques 21
  - 2.3.2 Specification of performability measures 22
  - 2.3.3 Composition of high-level model descriptions 23
  - 2.3.4 Mapping of high-level models to MRMs 24
- 2.4 Non-compositional state graph construction 24
- 2.5 Compositional state graph construction 25
  - 2.5.1 Fundamentals 26
  - 2.5.2 SG composition for pure interleaving 26
  - 2.5.3 SG composition for activity synchronization 27
  - 2.5.4 SG composition for sharing of SVs 28
  - 2.5.5 SG composition for replication of submodels 29
  - 2.5.6 Limitation of Kronecker operator driven composition schemes 29

3 Zero-suppressed Multi-terminal BDDs: Concepts, Algorithms and Application 31
- 3.1 Organization of the chapter 31
- 3.2 Binary Decision Diagrams and extensions 32
  - 3.2.1 Binary Decision Diagrams (BDDs) 32
  - 3.2.2 Zero-suppressed BDDs (z-BDDs) 37
  - 3.2.3 Multi-terminal BDDs (ADDs) 38

3.2.4 Zero-suppressed Multi-terminal BDDs (ZDDs) 39
3.3 Partially shared ZDDs (*p*ZDDs) 40
3.3.1 Definitions 41
3.3.2 Canonicity of *p*ZDDs 43
3.4 Operations on *p*ZDDs 45
3.4.1 Preliminaries 45
3.4.2 Applying binary operators to *p*ZDDs 46
3.4.3 Variants of the *p*`ZApply`-algorithm 52
3.4.4 Relabeling of variables 54
3.4.5 The *p*`ZRestrict`-operator 54
3.4.6 The *p*`ZAbstract`-operator 54
3.5 Applications 56
3.5.1 ZDD based representations of sets and relations 57
3.5.2 ZDD based representations of matrices 59
3.5.3 Extending ZDDs for efficiently computing matrix-vector products 66
3.5.4 Beyond DD based matrix representations 69
3.6 Related work and own contributions 69

4 The Activity/Reward-local Scheme: Symbolic SG based Analysis of High-level Markov Reward Models 73
4.1 Organization of the chapter 73
4.2 Model world 74
4.2.1 Static properties 74
4.2.2 Dynamic properties 75
4.2.3 Derived properties 81
4.2.4 Boundness of models 86
4.3 The activity-local scheme: Generating symbolic representations of state graphs 86
4.3.1 Main routine 86
4.3.2 Explicit state graph generation and encoding 88
4.3.3 Symbolic state graph composition 89
4.3.4 Symbolic reachability analysis 89
4.3.5 Re-initialization of the scheme 91
4.3.6 Example 91
4.4 Completeness and correctness of the scheme 94
4.4.1 Generation scheme 94
4.4.2 Composition scheme 96
4.4.3 Reachability analysis 98
4.5 Computing performability measures 99
4.5.1 Computing state probabilities 100
4.5.2 The reward-local scheme: Generating symbolic representations of reward functions 100
4.5.3 Computing moments of performance variables 102
4.6 Extending the basic activity-local scheme 103
4.6.1 Handling explicitly modeled symmetries 103
4.6.2 Handling of immediate activities 106
4.7 Related work and own contributions 108
4.7.1 Fully symbolic techniques 111
4.7.2 Semi-symbolic techniques 112
4.7.3 Semi-symbolic, compositional and submodel-interdependent techniques 114
4.7.4 Symbolic algorithms for generating the set of reachable states 116
4.8 Pre-published material 117

5 Empirics . . . 119
5.1 Organization of the chapter . . . 119
5.2 Preliminaries . . . 120
5.2.1 Employed models for benchmarking . . . 121
5.2.2 Layout of presented run-time data . . . 124
5.2.3 Platform . . . 124
5.2.4 Comparisons . . . 124
5.3 Assessing the activity-local SG generation scheme . . . 126
5.3.1 Comparing ADD and ZDD based SG generators . . . 126
5.3.2 Assessment of the new symbolic reachability analysis algorithm . . . 128
5.3.3 Significance of variable ordering . . . 130
5.4 Comparison of symbolic SG generation techniques . . . 131
5.4.1 Comparison to fully symbolic methods . . . 131
5.4.2 Comparison to semi-symbolic methods . . . 137
5.5 Assessing the ZDD based solvers . . . 142
5.5.1 Comparing ADD and ZDD based numerical solvers . . . 142
5.5.2 Choice of block and sparse level . . . 144
5.5.3 Significance of variable orderings . . . 146
5.6 Comparison with other solvers . . . 147
5.6.1 Comparison to the sparse matrix solvers of Möbius . . . 147
5.6.2 Comparison with the solvers of Smart . . . 149
5.7 Case Study: Telecommunication service system . . . 151
5.7.1 System description . . . 151
5.7.2 Model evaluation . . . 152
5.8 Pre-published material . . . 154

6 Conclusion . . . 155
6.1 Summary . . . 155
6.1.1 Zero-suppressed multi-terminal BDDs . . . 155
6.1.2 Activity/Reward-local scheme . . . 156
6.1.3 Computation of state probabilities . . . 157
6.1.4 Computing performability measures . . . 157
6.2 Benefits of *p*ZDDs and the activity/reward-local scheme . . . 158
6.3 Future work . . . 159

A Appendix: Mathematical Background . . . 161
A.1 Boolean functions . . . 161
A.2 Pseudo-boolean functions . . . 162
A.3 Kronecker operators . . . 162
A.4 Notation of modus ponens . . . 163
A.5 Pseudo-code and related notation . . . 163

B Appendix: Algorithms for BDDs and derivatives . . . 164

C Appendix: Algorithms for handling models with immediate activities . . . 165

References . . . 167

German Translations . . . 173

List of Figures

1 Introduction
1.1 Scheme for classifying the approaches to the state space explosion problem .. 3

2 Background Material
2.1 Bisimilar SGs 19
2.2 Compositional SG construction for interleaving and synchronization 26

3 Zero-suppressed multi-terminal binary decision diagrams
3.1 BDTs and the merging of isomorphic structures 34
3.2 *iso-free* BDD based representations of boolean functions 35
3.3 Shared or multi-rooted *iso-free* BDDs 36
3.4 *dnc*-free BDD based representations of boolean functions 37
3.5 *z*-BDD based representations of boolean functions 38
3.6 ADD based representations of pseudo-boolean functions 39
3.7 ZDD based representations of pseudo-boolean functions 40
3.8 *p*ZDD based representation of boolean functions 42
3.9 Function tables and *p*ZDD based representation 44
3.10 Resulting *p*ZDD and call-tree for $\mathsf{M} := \mathsf{A} \cdot \mathsf{B}$ 54
3.11 *sLTS*, its ZDD based representation and underlying transition rate matrix ... 60
3.12 *Mt*-DD based representation of the identity function $\mathbb{1}(\mathcal{V})$ 61
3.13 Block-wise access by dfs-traversal 62
3.14 Cross-product building and variable orderings ($\mathsf{M} := \mathsf{A} \times \mathsf{B}$) 64
3.15 Development and inheritance of concepts within BDT based data types 70

4 The Activity/Reward-local Scheme
4.1 A SPN and its underlying *sLTS* 92
4.2 Activity-local structures and binary encodings 93
4.3 Symbolic representation of the set of reachable states and the *sLTS* 93
4.4 Exemplification of the reward-local approach 102
4.5 SPN with user-defined symmetric submodels 104
4.6 Classification of symbolic SG generation methods 109

5 Empirics
5.1 Illustration of the adjunct processor (board) system [GLW00] 151
5.2 Single sub-unit specified as SAN 152

List of Algorithms

3 Zero-suppressed multi-terminal binary decision diagrams
3.1 Function for allocating unique *p*ZDD nodes only . . . 43
3.2 The generic *p*`ZApply`-algorithm . . . 49
3.3 *p*ZDD `op`-functions for boolean operators . . . 50
3.4 *p*ZDD `op`-functions for arithmetic operators . . . 51
3.5 The *p*`ZAnd`-algorithm implementing conjunction and multiplication . . . 53
3.6 The *p*`ZRestrict`-algorithm . . . 55
3.7 The *p*`ZAbstract`-algorithm . . . 56
3.8 Generating symbolic representations of singletons . . . 57

4 The Activity/Reward-local Scheme
4.1 Main routine for the activity-local SG generation scheme . . . 87
4.2 Procedures for explicit SG generation and encoding . . . 88
4.3 Variants of symbolic reachability analysis . . . 90
4.4 Re-initialization of explicit SG exploration and encoding . . . 91
4.5 Main routine for computing user-defined PVs . . . 100
4.6 Generating symbolic representations of reward functions . . . 101
4.7 Algorithm for computing moments of PVs via graph-traversal . . . 103
4.8 Applying the lumping theorem in case of user-defined model-symmetries . . . 105

B Appendix: Algorithms for BDDs and derivatives
B.1 The `Satisfy`-algorithm for BDTs and BDDs . . . 164
B.2 The `Satisfy`-algorithm for *z*-BDDs . . . 164
B.3 The `Satisfy`-algorithm for *p*ZDDs . . . 164

C Appendix: Algorithms for handling models with immediate activities
C.1 Explicit SG exploration in the presence of immediate activities . . . 165
C.2 Encoding state-to-state-transitions and testing for further exploration . . . 165
C.3 Symbolic reachability analysis for models with immediate activities . . . 166
C.4 Re-initialization when immediate activities are present . . . 166

List of Tables

5 Empirics

5.1 Model specific data for the various case studies 121
5.2 Run-time data of the activity-local scheme employing ADDs and ZDDs 127
5.3 Ratios for comparing ADDs and ZDDs 128
5.4 Comparison of the two variants of symbolic reachability analysis as implemented within the tool Möbius and by employing ZDDs 129
5.5 Comparison of the two variants of symbolic reachability analysis as implemented within the tool Caspa 130
5.6 Assessing the significance of the variable orderings 131
5.7 Comparing the activity-local scheme to Caspa 132
5.8 Comparing the activity-local scheme to Prism 135
5.9 Comparison to a non-compositional semi-symbolic SG gen. scheme 137
5.10 Comparison to the approach of [DKS03] (run-time data) 138
5.11 Comparison to the approach of [DKS03] (ratios) 140
5.12 Comparing activity-local scheme and Smart (run-times) 141
5.13 Comparing activity-local scheme and Smart (memory consumption) 142
5.14 ADD and ZDD based solution of CTMCs 143
5.15 HO ZDD based solution for different sparse and block levels 144
5.16 HO ZDD based solution for different variable orderings 146
5.17 Comparison with Möbius' sparse-matrix based solvers 148
5.18 Comparison with Smart's solvers: Run-time data for the Kanban model 149
5.19 Comparison with Smart's solvers: Run-time data for the FMS model 150
5.20 Data as obtained for analyzing the case study 153

1

Introduction

1.1 Motivation

It is commonplace that complex hard- and software systems have become part of our daily life. Because of our high dependency on these systems, it becomes more and more important to assert that they are working correctly and that they meet high requirements concerning performance and dependability. However, practice may forbid to directly obtain the data for evaluating a system's performance and/or dependability, commonly denoted as performability. In such cases, where the system under study is not directly accessible to carry out system tests or system measurements, one is restricted to analyze a (mathematical) system model instead. The major advantage of such a (formal) procedure is obvious: The model-based evaluation enables one to assess the functionality and the quantitative behavior of a not necessarily existing system, so that one is already capable to assert the correctness and dependability of a system design in the early stage of the (re-)design process and thus may avoid costly maldevelopment.

Annotated state/transition-systems (ST systems) give an adequate framework for formally describing complex system behaviors. However, nowadays hard- and software systems are often parallel or even distributed, resulting in a high degree of complexity, so that a detailed system description as a ST system is often not only hampered, but simply impossible due to the size of the resulting model. Formal high-level model description methods, as developed in the past decades, have shown to be powerful tools for compactly describing complex systems. By including a stochastic concept of time and costs and/or gains into the model description, one obtains what is denoted as high-level stochastic performability model. Depending on the high-level model, on the employed formalism, and on the probability distributions used for describing the timed delay between successive system states, one either may evaluate the desired performability of the system analytically, numerically or empirically. Obtaining a solution analytically, i.e. via evaluating a closed-form expression, is in principle restricted to a limited class of queuing systems. For empirically or numerically evaluating a system model's performability measures, one is forced to partially or completely generate and analyze a model's underlying annotated ST system. In contrast to empirical model evaluation, as provided by simulation studies, where only traces of a system's behavior are generated, high-level Markov reward models allow their exhaustive analysis, which requires the complete generation of the underlying ST system. However, the benefit of an exhaustive analysis comes at the drawback that the system behavior is required to be Markovian. I.e. timely delayed state changes are only allowed to occur after the elapsing of a time span, the length of which is described by an exponentially distributed random variable.

Markov reward models considered in this thesis are compactly described by means of Markovian extensions of well-known high-level model description formalisms, such as generalized stochastic Petri nets (GSPNs), stochastic activity networks (SANs) and stochastic process algebras (SPAs), to name only a few of them. In order to analyze them, the high-level model description is transformed into a finite, stochastic ST system, also often denoted as low-level representation or (stochastic) state graph (SG). The ST system can directly be interpreted as Markov reward model (in a mathematical sense). The theory of models of this kind is well-known. It allows one to numerically compute a probability distribution on the set of system states, where for evaluating the desired performability measures these state probabilities are aggregated.

1.2 State space explosion problem and related approaches

The first step when analyzing a high-level Markov reward model is the generation of the low-level model representation. Here one already faces the notorious *state space explosion problem*

State space explosion

The concurrency of activities must be made explicit when transforming high-level models into their low-level representations. The interleaving semantics of standard Markovian model formalisms yields an explicit extraction of all possible execution sequences of system activities, when generating the (stochastic) SG. Consequently, this may lead to an exponential blow-up of the SG in the number of system states. This phenomenon is commonly addressed as *state space explosion* problem. In the context of high-level Markov models the state space explosion problem hampers or may prevent the SG based analysis of complex and large systems for the following two reasons: (a) The number of system states is to large, so that they can not be kept in memory nor is their individual generation feasible. (b) The transition rate matrix of the high-level model underlying SG imposes a non-tolerable memory requirement.

Before discussing recent approaches for coping with this two-fold problem, the traditional technique for analyzing high-level Markov reward models will be discussed, where its limitation as imposed by the state space explosion problem is emphasized.

Traditional exploration techniques

The traditional technique of generating all reachable states of a modeled system is called *exhaustive* state space exploration, where the individual visiting of states is denoted as *explicit* exploration in the following. The states, which can be visited by starting at the initial system state, give the set of reachable states. The data structure required for generating the complete set of reachable system states is a buffer (state buffer) and a large somehow structured storage space (state table). The latter holds the detected states, where the former holds the detected, but yet not explored ones. Usually a state is considered to be explored once all its successor states have been determined. Since the state buffer is accessed in a structured manner, the currently not used parts can be swapped onto secondary storage device. Thus and contrary to the state table, the state buffer is not the bottleneck of an exhaustive and explicit state space exploration. The state table serves as a database, its single purpose is to determine if a reached state is already known or not. Since the state table is accessed in an unstructured way, and hard drive access is computational expensive, the number of system states to be explored is restricted by the size of the available random access memory (RAM). As a consequence the the size and complexity of high-level models and thus systems to be analyzed is strongly restricted in practice.
For exemplification one may think now of a SG, where each state consists of 10^3 counters. Let the values of the counters be bounded to 255, so that each state descriptor consumes approx. $0.977\,KByte$. Thus one already requires approx. $0.93\,GByte$ for storing 10^6 distinct states. But not enough, the look-up of states and their storage into memory, which includes hashing and collision resolution, as well as the explicit generation of all transitions among the systems states, induces a not ignorable run-time overhead. So let us assume that there is enough RAM available, let us say for storing 10^8 states, at an average of $1.3E{-}4\,secs$[1] of CPU time spent for the processing of each state, one is already forced to wait approx. $3.6\,h$ for transforming a high-level model consisting of 10^8 system states into its low-level representation.

[1] This is the average CPU time consumed per state by the SPN based tool DSPNexpress for generating and storing approx. 10^6 system states, where the tool ran on a 64-bit AMD Opteron architecture.

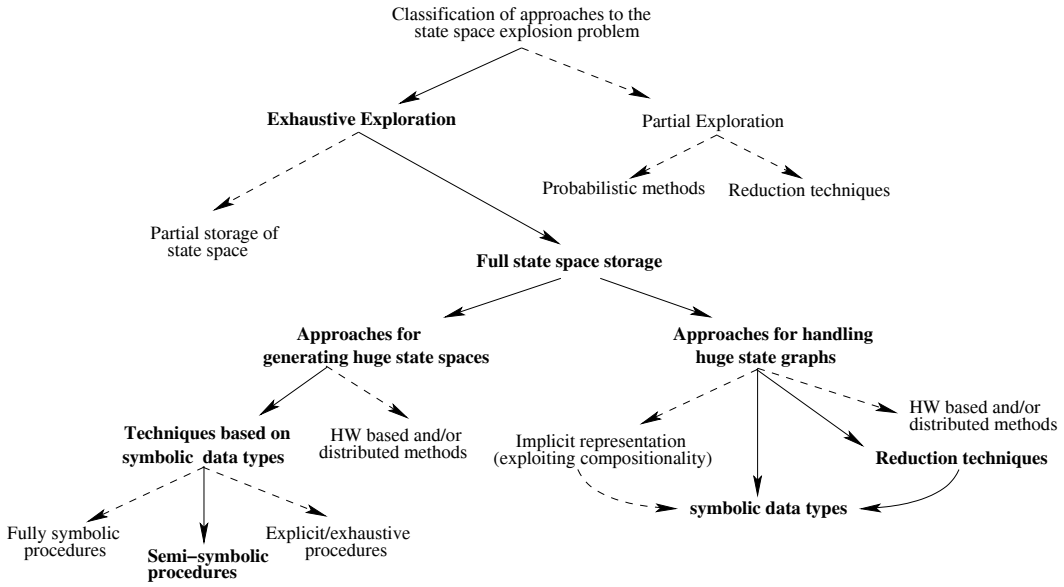


Figure 1.1: Scheme for classifying the approaches to the state space explosion problem

Traditional SG storage techniques

Once all system states and the transitions among them are generated, one numerically computes the individual state probabilities. This is done on the basis of the transition matrix, as to be derived from the generated SG. The matrix is commonly stored in a memory efficient "sparse matrix format", but nevertheless, its size may impose a non ignorable memory requirement E.g. the SGs of the well-known "Kanban Manufacturing System" and "Flexible Manufacturing System" benchmark models, (cf. Chapter 5), the SGs of which consist of $\sim 2.5E6$ and $\sim 4.5E6$ system states, already require ~ 380 and ~ 500 MBytes, when storing the respective transition rate matrix in "sparse matrix format".[2]

For the above reasons the traditional technique is currently restricted to analyze models consisting of clearly less than 10^7 system states on a commodity computer.

Classification of approaches

For coping with the state space explosion problem on the one hand and the limited availability of memory space and CPU time on the other hand, many approaches have been developed. A classification of existing methods is illustrated in Fig 1.1, where we concentrated on approaches developed in the context of high-level Markov models. At the top level one may distinguish between approaches that perform partial state space exploration and approaches executing exhaustive state space search. There exist many ways to organize a partial state space exploration. The two most prominent representatives of this class are reduction techniques and probabilistic methods. Reduction techniques aim to prune away redundant activity-sequences. This can be achieved, for example, by defining an equivalence relation on the system behavior (e.g. [God95]) or exploiting user-defined symmetries within the high-level model specification, so that one is enabled to apply the state lumping theorem on-the-fly (cf. Sec. 2.4, p. 24ff). Using probabilistic methods, large state space can be stored. But due to not resolving hash collisions different states my be falsely considered as identical, and thus only a fraction of all reachable states may be generated. As a consequence the probability that states are omitted, and thus the probability that the computed performability

[2] The values are the ones as obtained when analyzing the two benchmark models with the Möbius modeling tool [DCC+02] and its standard Markov reward model analysis module, details will follow in Chapter 5, cf. Tab. 5.1 (p. 121) and Tab. 5.17.B (p. 148).

measures are incorrect, is greater than zero.[3]

The approaches that perform exhaustive state space search, can be divided into two classes:

(1) Approaches that store a reduced SG: This is achieved by not (permanently) storing not needed states during exploration, e.g. by eliminating vanishing states on-the-fly.

(2) Concerning the exhaustive approaches, which store the complete state space, one may now differentiate between methods that target the exploration of huge state spaces and between methods, which are concerned with the storage of huge SGs, as well as the efficient computation of performability measures.

(2.a) Approaches for generating huge state spaces: An exhaustive state space generation can be organized by

i. making use of powerful hardware, i.e. utilizing mass storage and/or distributed hardware (e.g. [Kno99, HW06]), or

ii. making use of symbolic SG generation techniques (cf. Sec. 1.3).

(2.b) Existing techniques for storing and handling huge SGs can be grouped into the following four classes:

i. Methods employing mass storage and/or making use of distributed hardware, (e.g. [Kno99, HBB99, Meh04]).

ii. Methods making use of reduction techniques a posteriori to SG generation by detecting and exploiting an equivalence relation on the system behavior (cf. Sec. 2.2.3, p. 17ff).

iii. Implicitly representing the SG with the help of Kronecker operators (e.g. [Pla85, Buc91, Sie95, CT96]).

iv. Methods making use of symbolic data types (Decision diagrams) (cf. Sec. 1.3).

As illustrated in Fig. 1.1 the employment of symbolic data types (Decision diagrams) yields the nice feature that they not only support an efficient exploration of huge SGs, but also realize a compact storage of the SG, its transition rate matrix resp.. Furthermore, other techniques may make use of them for increasing their efficiency, for exemplification one may think of implicit (symbolic) matrix representation techniques [Sie98, CM99b], decision diagram based SG reduction techniques [Sie02] among others. Concerning the above made classification, the contribution of this thesis to the alleviation of the state space explosion problem on the basis of a (new) symbolic data type is therefore three-fold: (a) A method for efficiently generating the symbolic representation of a high-level Markov models SG will be introduced. (b) An approach for efficiently storing and handling the obtained SG will be developed, so that performability measures of the system under study can be computed efficiently. (c) We will also present an approach, which exploit user-defined model symmetries and delivers a reduced SG, so that the numerical analysis can operate on a reduced number of system states. Thus the symbolic framework, as to be developed in this thesis, is to be characterized by the bold-faced concepts contained within the classification of Fig. 1.1.

1.3 State-of-the symbolic techniques

Decision diagrams can be employed for efficiently storing function tables. Consequently it is straight forward to employ them for representing characteristic functions of finite sets and thus for representing sets of states and/or transition relations. Approaches making use

[3] An overview over probabilistic methods can be found in [KL04].

of such a storage scheme are commonly denoted as symbolic state space representation techniques.

Symbolic data types

The usage of Binary Decision Diagrams (BDDs) in today's CAD-tools is state-of-the technique, since they are known to be extremely efficient in the representation of boolean functions and thus highly suited for representing the characteristic functions of sets. Furthermore, efficient algorithms for their manipulation are known [Bry86]. Throughout the last decade many derivatives have been developed in order to employ them successfully not only in the context of hardware verification, but also in the context of applications, where extremely large sets of number strings are needed to be kept in the RAM, e.g. [SS03]. Thus, it is not surprising that the field of stochastic modeling has taken advantage of this kind of symbolic set or symbolic transition relation representation. In the context of stochastic modeling, the most prominent decision diagrams are *multi-terminal* or *algebraic BDDs* (ADDs) [FM97], *multi-valued decision diagrams* (MDDs) [KVBSV98] and *matrix diagrams* (MxD) [Min01].

Since ADDs are the multi-terminal extension of BDDs the most important BDD-algorithms are directly applicable to them and many implementations exist. Consequently this type of decision diagram has a long history, when it comes to the modeling of systems. However, in the context of high-level model description, it turned out that the BDD-specific *don't care* reduction rule is from minor interest for the memory-efficiency (as far as paths leading to the terminal 1-node are concerned) [Par02].

Generation techniques

Techniques for generating a symbolic representation of a high-level model's underlying SG range from the explicit generation and encoding of all states (exhaustive) [Web02, DKK02], up to fully symbolic approaches [PC98, KS02, AKN$^+$00] (cf. Fig. 1.1). In case of the latter a symbolic representation is even directly derived from the high-level model description making the fully symbolic approaches highly computational efficient. However, contrary to methods making use of conventional SG exploration, the fully symbolic methods require the high-level model formalism to possess a symbolic semantic. Another important class of approaches, which in contrast to the fully symbolic methods is independent of the employed modeling formalism, are the so-called semi-symbolic, compositional techniques [CM99b, CLS01, HMKS99, Sie01]. Techniques of this kind are characterized by a combination of explicit exploration and purely symbolic manipulations, where contrary to explicit exhaustive methods, a composition scheme is employed.[4] Compositionality seems to be crucial, not only for the semi-symbolic methods, since it not only reduces the runtime, as not all sequences of independent activities have to be extracted explicitly, but also induces regularity on the symbolic structures and thus reduces the peak memory consumption of the schemes.

However, the symbolic SG generation techniques mentioned above are limited to cases where

(1) the high-level formalism is of a specific kind [PC98, KS02, AKN$^+$00],

(2) bounds of components of the state descriptor are either specified directly in the high-level model [KS02, AKN$^+$00], or can be computed by means of an invariant analysis [PRCB94, DKK02], or where

(3) the high-level model possesses a compositional or a specific decomposable structure and the SGs of the submodels can be generated in isolation [CM99b, HMKS99, AKN$^+$00, Sie01, CLS01, KS02, LS02].

[4] Applying a composition scheme means, that the SG of the overall model is constructed from smaller components, commonly from the SGs of the user-defined submodels (submodel-local SGs), where details will follow in chapter 2.

Recently developed semi-symbolic, compositional methods as presented in [CMS03, DKS03] generate the submodel-local SGs in a submodel-interdependent fashion, overcoming the above restrictions. But nevertheless their efficient application is still restricted, for the following reasons:

(1) The technique applied in [CMS03] requires a decomposition of the high-level model into (independent) partitions, in such a way that the overall model's SG can be obtained by making use of a Kronecker operator driven composition scheme (cf. Sec. 2.5, p. 25ff).

(2) The technique employed in [DKS03] makes use of the user-defined modular structure of the overall model, so that inefficiency occurs, if the interaction among the submodels is not very limited.

(3) Concurrency taking place within the submodels is not exploited in general. Consequently, shuffled sequences of independent activities are fully expanded at the submodel level, so that in case of submodels with extremely large SGs, the methods are in general not very efficient.

The above discussion leads to the following focal aims for a new scheme:

- Independence of the employed high-level model description technique.
- The individual treatment of states (both their exploration and encoding) should be avoided as much as possible.
- The scheme should be applicable to both, to composed and non-composed models, i.e. to models possessing a modular structure as well as to models which are non-modularly (monolithically) structured.
- But on the level of SGs, it should in any case exploit some form of compositionality.

SG reduction techniques

Even under the symbolic techniques the number of system states, the probabilities of which must be computed, is the bottle-neck of SG based analysis of high-level Markov reward models. Sometimes it is possible to reduce the SG, so that one only needs to compute the individual state probabilities for a smaller number of states. The following classes of approaches exists:

(1) High-level model reduction techniques:
By applying transformation rules, which depends on the employed high-level formalism, a high-level model may be transformed into a simpler one, exhibiting the same timed behavior concerning the performability measures to be computed. At exploration this transformed model may yield a smaller SG if compared to the original model.

(2) On-the-fly: These techniques are applied during SG generation, allowing to assess models, the analysis of which would otherwise be impossible.

(3) A-posteriori to SG generation: These techniques are applied to the complete SG. Contrary to the on-the-fly strategies, the techniques of this class are much more general. For exemplification one may think of applying the lumping theorem or eliminating time-less traps within the system behavior.

Due to the above mentioned reasons the development of SG reduction techniques is still a vibrant field of research activity, where respective symbolic approaches have been developed in the past years. Unfortunately the symbolic algorithms, as well as their explicit counterparts, are known to be computational expensive.

Numerical solution techniques

Once a symbolic representation of a high-level model's underlying SG is generated, the next thing to do is the computation of state probabilities. However, in the context of symbolic SG representations, only the employment of iterative numerical solution methods seems useful. As known from practice, fully symbolic methods, which besides a symbolically represented generator matrix also employ symbolically represented iteration vectors, are not very efficient [Fra99]. Therefore the hybrid solution technique [Par02] is currently well established in practice. Within a hybrid implementation of a numerical solution method, the transition rate or generator matrix is stored symbolically and the iteration vectors are stored as arrays, which speeds-up the iteration-steps significantly. After the state probabilities are computed, the next thing to do, is the evaluation of user-defined performability measures by computing rate- and impulse reward values of states and aggregate them accordingly.

1.4 Contributions of this thesis

In this thesis, we present a symbolic framework for the analysis of very large (finite) continuous-time Markov Reward models. For representing the Markov reward models, a new type of decision diagram is introduced, which we denote as zero-suppressed multi-terminal binary decision diagram (ZDD). For working with ZDDs, this work also develops a set of symbolic algorithms. This new type of symbolic data structure is then employed within a new (semi-symbolic) scheme for efficiently generating a symbolic representation of a Markov Reward model as stemming from a high-level model description. This scheme, which we denote as activity/reward-local scheme is based on partial explicit SG exploration, a new scheme for symbolic composition and a new scheme for symbolic reachability analysis. In contrast to existing techniques, it employs model-inherent structures, rather than explicitly user-defined ones. Consequently, the activity/reward-local scheme is highly suited for being applied not only for compositional model worlds, but also for monolithic ones; i.e. to be applied in cases where models cannot be decomposed in a Kronecker operator compliant way (cf. Sec. 2.5.6, p. 29f) or if individual submodel-local SGs are disproportionally large.

In order to reduce the number of states, the probability of which must be computed, we augment the basic activity/reward-local scheme with a symbolic algorithm, which executes a SG reduction, when user-defined symmetries are present.

For computing state probabilities, we extended the hybrid solution method, as known from other symbolic approaches, to the case of the here newly introduced ZDDs. Since we also handle user-defined performability measures, the individual contributions of this thesis yield a complete approach for the SG based analysis of high-level Markov Reward models with very large underlying state/transition systems.

For evaluating the applicability of the developed concepts, an implementation has been integrated into the multi-formalism performance evaluation tool Möbius [DCC+02]. This allows not only the analysis of well-known benchmark models, but also to evaluate the availability of an adjunct processor system as employed in the telecommunication industry, so that the performance of the here presented approach could be assessed.

1.5 Organization of the thesis

This thesis is organized as follows: Chapter 2 recapitulates the required background knowledge and gives needed definitions (see also Appendix A). Chapter 3 introduces our new type of decision diagram, as well as the important algorithms for its efficient manipulation. A survey on ZDD based set and matrix representation, as well as an introduction to the ZDD-based hybrid solution methods for solving systems of linear and differential equations will round the chapter. Chapter 4 explains our new semi-symbolic, compositional approach for the efficient SG based analysis of high-level Markov reward models. Empirical results,

including comparisons with other tools and implementations, are presented in Chapter 5. Chapter 6 concludes the thesis by presenting a summary and indicating future steps.

2

Background Material

2.1 Organization of the chapter

For carrying out qualitative and quantitative analysis of systems it is required to specify a mathematical model representing the system under study. Markovian models build the fundament for a wide range of different techniques for evaluating systems. Sec. 2.2 recapitulates aspects concerning the theory of Markovian models, including the definition of continuous-time Markov Reward models (MRMs), approaches for their numerical solution, basics about reduction of MRMs and different types of state/transition systems, as commonly employed for representing MRMs. Complexity of todays hard- and software systems advises the use of a formal high-level modeling technique, rather than specifying the system to be analyzed directly as a stochastic state/transition system.
Sec. 2.3 introduces the basic high-level modeling techniques, discusses how performability measures can be defined and briefly introduces methods for organizing high-level models in a hierarchic and compositional style. In total, these methods allow one to define what we denote as a high-level MRM. Via state graph (SG) generation, as already mentioned in Sec. 1.2 (p. 2f), a high-level MRM can be mapped to a ST system, which itself can be interpreted as MRM and therefore in the following as low-level MRM referred to. If the high-level model possesses a compositional structure, this structure can be exploited in the process of SG generation, so that the overall SG of the high-level model can be constructed from smaller components. How high-level models can be mapped to MRMs via SG generation is briefly explained in section 2.3.4.
However, since this thesis develops a new scheme for carrying out SG generation, a detailed discussion on this issue needs to follow. This is done in Sec. 2.4 for monolithic or non-compositional SG generation procedures and in Sec. 2.5 for the compositional ones. Since compositionality is essential for symbolic SG representation techniques, at least for the ones relying on (partial) explicit SG exploration, Sec. 2.5 introduces basic methods for compositionally constructing SGs. The well known Kronecker operator driven composition schemes are discussed there, as well as a new scheme for composing submodels interacting via the sharing of variables.

Some mathematical basics about boolean functions and their expansion as directly employed in this thesis are briefly re-visited in the appendix (cf. Appendix A). Furthermore it also defines the Kronecker product (KP) and the Kronecker sum (KS) of matrices and clarifies the pseudo-code and related definitions as employed in this thesis.

2.2 Markov Theory

In the following we introduce some basic concepts of Markov theory.

2.2.1 Continuous-time Markov reward model (MRM)

In the context of this thesis a system is specified by making use of a high-level modeling technique, where each specified model can ultimately be mapped to a (low-level) Markov Reward model. In the following the discussion is limited to models of this kind. However, due to the nature of the high-level model description methods, it appears that after SG generation one may end up with a SG, where two kinds of transitions between pair of states exists, namely timed and instantaneous transitions. However, as it will be discussed in Sec. 2.2.3, the latter can be eliminated, so that one finally ends up with a SG which defines

a (pure) continuous-time Markov chain (CTMC) For this reason, we directly concentrate on CTMCs in the following paragraphs.

Continuous-time Markov Chain (CTMC)

A stochastic process is a family of random variables $\{X(t)|\, t \in N\}$. Parameter t is commonly associated with the system time. The co-domain of $X(t)$ is called state space. In the context of this thesis the latter is assumed to be a finite set of vectors of constant dimension ($\vec{s} \in \mathbb{S}$). However, for simplicity we will employ simple state indices like i and j in the paragraphs to follow, rather than making use of a vector-oriented notation as employed latter. Stochastic processes with discrete state spaces are commonly denoted as chains.
The property of being memoryless gives for a process, that its future evolution depends only on the current state and not on the past history. Chains possessing this property are known as Markov chains.

Definition 2.1: Markov chain

A state discrete stochastic process is denoted as Markov chain if the following holds: $\forall n \in \mathbb{N}$ and $\forall t_0 \leq t_1 \leq \cdots \leq t_{n-1} \leq t_n$ it holds that

$$\begin{aligned} Prob[X(t_n) = j|\, X(t_{n-1}) = i_{n-1}, X(t_{n-2}) = i_{n-2}, \ldots, X(t_0) = i_0] \\ = Prob[X(t_n) = j|\, X(t_{n-1}) = i_{n-1}] \end{aligned}$$

The above property implies that a Markov process is not only independent of the sequence of visited states in the past, but also that its sojourn time T_i to be spent in the current state i is independent of the sojourn time already elapsed:

$$Prob(T_i > t + \triangle t | T_i > t) = Prob(T_i > \triangle t)$$

The only discrete probability distribution which satisfies this property for modeling discrete sojourn times is the geometric distribution: $Prob(T_i = n) = p_{i,i}^{n-1}(1 - p_{i,i})$, where T_i is the random variable which gives the number of time steps the Markov chain stays in state i and where $p_{i,i}$ is the (discrete) one-step probability of staying in state i. Randomly distributed continuous sojourn times, satisfying this property can only be modeled with the exponential distribution: $Prob(T_i \leq t) = 1 - e^{-\lambda_i t}$. Whether parameter t is discrete (number of steps) or continuous, one speaks of discrete-time Markov chains (DTMC) or continuous-time Markov chains (CTMC) [Ste94]. DTMC or CTMC are commonly defined by a set of states, a matrix of transition probabilities or transition rates and a initial probability distribution on the set of states. Since DTMCs are of minor interest in the context of this work, we solely concentrate on CTMCs which the define as follows:

Definition 2.2: Continuous-time Markov Chain

A continuous-time Markov chain (CTMC) is defined by a triple $C := (\mathbb{S}, T, \vec{\pi}(0))$. $\mathbb{S}$ is the finite set of system states and T is the matrix of transition rates among the states. I.e. T gives a mapping $\mathbb{S} \times \mathbb{S} \mapsto \mathbb{R}_0^+$ where $\nexists i \in \mathbb{S} : T(i,i) \neq 0$ holds. Vector $\vec{\pi}(0)$ defines an initial probability distribution on $\mathbb{S}$.

In the context of this thesis, we are only concerned with time-homogenous CTMCs, i.e. with CTMCs where the transition rates are time-independent. Cycles within a Markov Chain which can not be left are denoted as bottom strongly connected components (BSCC), i.e. for a given state i of the BSCC each other state j of the BSCC can be visited by taking a finite number of intermediate transitions. States of a Markov chain which do not belong to a BSCC are denoted as transient. In this sense an absorbing state is a BSCC consisting of a single state. A Markov chain consisting of a single BSCC is denoted as irreducible. In the

following we are solely concerned with finite irreducible CTMCs. How to handle CTMCs with absorbing states and/or non-irreducible CTMCs can be found in the literature [Ste94].

Markov Reward Models (MRM)
A MRM consists of a CTMC and a set of reward functions, where two kinds of rewards are known: (a) rate rewards, which are associated with states and (b) impulse rewards, which refer to transitions. In the context of this work, the specific state- and/or transition-dependent reward values are assumed to be time-independent and we define them as follows:

Definition 2.3: Rate reward of a CTMC

A rate reward r defined on a CTMC is a function $\mathcal{R}_r : \mathbb{S} \to \mathbb{R}$

The set of all rate rewards defined for a given CTMC is denoted $\mathcal{R}$.

Definition 2.4: Impulse reward of a CTMC

An impulse reward a of a CTMC is a function $\mathcal{I}^a : \mathbb{S} \times \mathbb{S} \to \mathbb{R}$.

The set of all impulse rewards defined for a given CTMC is denoted $\mathcal{I}$

Based on the above definitions one is now enabled to define a continuous-time Markov reward model as follows:

Definition 2.5: Continuous-time Markov Reward Model

A continuous-time Markov reward model (MRM) is defined by a triple $M := (C, \mathcal{I}, \mathcal{R})$, where C is a CTMC, $\mathcal{I}$ is a set of impulse reward functions and $\mathcal{R}$ a set of rate reward functions.

2.2.2 Numerical solution of MRM

For the CTMC one may compute a probability distribution on the set of system states, which of course evolves over time. For computing the actual rate and impulse rewards the state probabilities and reward functions are combined and somehow aggregated. At first we will now derive the system of equations to be solved for obtaining state probabilities. This will be followed by introducing methods for computing the probability distribution on the set of states of the CTMC, where such solution methods, i.e. the software implementing them is commonly denoted as numerical solver. After state probabilities are obtained, one is now in the position to compute rewards, which together with the probability distribution evolve over time.

Chapman-Kolomogoroff system of equations
It is known that the minimum of n exponentially distributed random variables is once again exponentially distributed, with parameter $\lambda := \sum_1^n \lambda_i$. The probability of random variable X_i "to win the race" and thus to hold the minimum value, is then given by $\frac{\lambda_i}{\sum_1^n \lambda_j}$. Within a CTMC $C := (\mathbb{S}, T, \vec{\pi}(0))$ and a state $i \in \mathbb{S}$ this situation is given if state i can be left via n (exponentially delayed) transitions. This gives the probability for moving from state i to state j within time interval $\triangle t$ as follows:

$$p_{i,j}(\triangle t) := \begin{cases} \frac{T(i,j)}{E(i)} \cdot \left(1 - e^{-E(i)\triangle t}\right) \Leftrightarrow E(i) \neq 0 \\ 0 \qquad\qquad\qquad\qquad \text{else} \end{cases} \tag{2.1}$$

where $E(i) := \sum_1^n T(i,j)$ is the sum of the i'th row, also commonly denoted as exit rate of state i. Rows with row sums equal to 0, i.e. holding 0-entries only, refer to absorbing states. It is intuitive clear that such states do not have outgoing transitions, thus the probability of leaving an absorbing state must be 0 and the probability of the process of staying in such a state, once it is reached, is 1. Since the above equation gives a probability for each pair of states $(i,j) \in \{\mathbb{S} \times \mathbb{S}\}$ it enables onto construct the matrix of transition probabilities $P(\triangle t)$.

Now we are interested in the marginal transition probabilities, i.e. the probabilities for $\triangle t \to 0$:

$$q_{i,j} := \lim_{\triangle t \to 0} \frac{p_{i,j}(t+\triangle t) - p_{i,j}(t)}{\triangle t}$$

where t can be set to 0, since we are only concerned with time-homogenous CTMCs. For computing this marginal probability, we need to differ among the cases $i \neq j$ and the case that the process remains in state i.

(1) A single transition from i to j occurred:
As t grows, the probability of leaving state i grows, but as t diminishes this probability falls towards 0. Thus the probability of a transition taking place in state i at time-point $t = 0$ is equal to 0. By employing this in Eq. 2.1 and L'Hospital's theorem it follows:

$$q_{i,j} := \lim_{\triangle t \to 0} \frac{p_{i,j}(\triangle t)}{\triangle t} = T(i,j) \,\text{for}\, i \neq j \tag{2.2}$$

(2) The process stays in its state:
As t grows, the probability of staying in state i falls, but as t diminishes the probability of staying in i grows towards 1, thus $p_{i,i}(0)$ must be equal to 1. Employing this in Eq. 2.1 yields:

$$q_{i,i}(\triangle t) := \lim_{\triangle t \to 0} \frac{p_{i,i}(\triangle t) - 1}{\triangle t}$$

Since $p_{i,i}(\triangle t) = 1 - \sum_{i \neq j} p_{i,j}(\triangle t)$ it follows:

$$\begin{aligned} q_{i,i}(\triangle t) &= \lim_{\triangle t \to 0} \frac{1 - \sum_{i \neq j} p_{i,j}(\triangle t) - 1}{\triangle t} \\ &= \lim_{\triangle t \to 0} \frac{-\sum_{i \neq j} p_{i,j}(\triangle t)}{\triangle t} \end{aligned}$$

which is something we already know, namely the negative exit rate of state i:

$$q_{i,i}(\triangle t) = -\sum_{i \neq j} T(i,j) = -E(i) \tag{2.3}$$

If one writes the results achieved above in a matrix notation, one obtains:

$$Q := \lim_{\triangle t \to 0} \frac{P(\triangle t) - \mathbb{1}_{\mathbb{S}}}{\triangle t}$$

where $\mathbb{1}_{\mathbb{S}}$ is a $\mathbb{S} \times \mathbb{S}$ identity matrix. Matrix Q is commonly denoted as (infinitesimal) generator matrix.

Based on the transition probability matrix $P(\triangle t)$ as constructed in Eq. 2.1, one is enabled to define the probability for being in state i at time-point $t + \triangle t$ as follows:

$$\vec{\pi}(t + \triangle t) = \vec{\pi}(t) P(\triangle t)$$

For computing state probabilities at arbitrary time points, one simply needs to compute the differential of t and $t + \triangle t$:

$$\frac{\delta \vec{\pi}(t)}{\delta t} = \lim_{\triangle t \to 0} \frac{\vec{\pi}(t + \triangle t) - \vec{\pi}(t)}{\triangle t} = \vec{\pi}(t) \lim_{\triangle t \to 0} \frac{P(\triangle t) - \mathbb{1}_{\mathbb{S}}}{\triangle t} \tag{2.4}$$

Due to the results achieved above $\lim_{\triangle t \to 0} \frac{P(\triangle t) - \mathbb{1}_{\mathbb{S}}}{\triangle t}$ is already known, it is the generator matrix Q. This gives us finally the well-known Chapman-Kolomogoroff system of differential equations:

$$\frac{\delta \vec{\pi}(t)}{\delta t} = \vec{\pi}(t)Q.$$

Its solution gives the distribution of probabilities on the set of states at time point t.

Computing state probabilities

For numerically computing state probabilities one needs to solve the Chapman-Kolomogoroff system of equations. If one is interested in the distribution of the state probabilities after a specific time-interval has elapsed, commonly denoted as transient analysis, one needs to solve a set of differential equations. Steady-state analysis, yields the distribution of states probabilities on the long-run, i.e. $t \to \infty$.

In the following the will briefly recapitulate the basic schemes as found in practice for computing transient and steady state probabilities for CTMCs. The interested reader may refer to the textbook [Ste94] for details.

(A) Transient analysis

A well-known approach to solve the Chapman-Kolomogoroff system of differential equations is based on the uniformization of the generator matrix. Via uniformization one discretizes a given CTMC, i.e. one transforms the CTMC into a DTMC. This is achieved as follows:

$$P := Q\triangle t + \mathbb{1}_{\mathbb{S}} \text{ with } \triangle t = \frac{1}{c \cdot \max\limits_{i \in \mathbb{S}} |q_{i,i}|} \text{ and where } \triangle t \leq \frac{1}{\max\limits_{i \in \mathbb{S}} |q_{i,i}|} \text{ must hold.}$$

The constant $u := c \cdot \max\limits_{\mathbb{S}}(|q_{i,i}|)$ is commonly denoted as uniformization constant, a good choice in practice is obtained for $c := 1.02$. Applying the closed-form solution for a system of differential equations, where Q of the Chapman-Kolomogoroff system is substituted with $u(P - \mathbb{1}_{\mathbb{S}})$ gives:

$$\vec{\pi}(t) = \vec{\pi}(0) \cdot e^{uP \cdot t} \cdot e^{(-u)t}.$$

For the Taylor expansion $e^x = \sum_{k:=0}^{\infty} x^k/k!$ one obtains

$$\vec{\pi}(t) = \sum_{k:=0}^{\infty} \vec{\pi}(0) P^k \frac{(ut)^k}{k!} \cdot e^{(-u)t}$$

This is the computation of distribution of state probabilities after k time steps of the discretize CTMC ($\vec{\pi}(0)P^k$) and a weighting of the obtained state probabilities with the discrete probabilities of a Poisson distribution for k arrivals ($p(k, u, t) := \frac{(ut)^k}{k!} \cdot e^{-ut}$). From the above equation it also arises that the state probabilities depend on the initial distribution of probabilities on $\mathbb{S}$. However, for a irreducible CTMC and a large value t, this dependency fades away. Fortunately it suffices to compute the above sum for the summation index set $L \ldots R$, rather than computing the infinite sum. The choice of the left- and right truncation point (L, R) depends on the product ut and a user-defined maximum round-off error. Details on the computation of the Poisson probabilities and the truncation points can be found in [FG88].

(B) Steady-state analysis

For irreducible CTMCs the state probabilities are independent of t and the differential quotient $\frac{\delta \vec{\pi}(t)}{\delta t} = 0$, as $\triangle t \to \infty$ (cf. Eq. 2.4). Contrary to the transient analysis, one therefore needs to solve solely a set of linear homogenous equations:

$$\vec{0} = \vec{\pi}Q$$

In contrast to irreducible CTMCs, for reducible CTMCs the distribution of steady-state probabilities depends on the initial distribution. I.e. one may be forced to compute the steady-state distribution of the different BSCCs individually and weight the obtained results with the respective probability of ending up in the respective BSCC.

Direct solution methods are computational expensive and since the coefficient matrix must be adapted after each step, iterative methods are more likely to be found in practice. In the following paragraphs we will briefly introduce the basic schemes as far as it is from concern for this work, for a detailed discussion the reader may once again refer to [Ste94].

*Power method (*POW *method):* The Power method is also based on uniformization, yielding the following iteration scheme:

$$\vec{\pi}^{k+1} = \vec{\pi}^{k} \cdot (Q \triangle t + 1_{\mathbb{S}}) \text{ with } \triangle t < \left(\max_{i \in \mathbb{S}} (|q_{i,i}|) \right)^{-1} \text{ must hold.}$$

To achieve good convergence $\frac{0.99}{\max_{i \in \mathbb{S}}(|q_{i,i}|)}$ is known to be a good value for $\triangle t$. In an element-wise notation we have:

$$\pi_i^{k+1} := \pi_i^k \sum_{j \in \mathbb{S}} \pi_j^k Q(j,i) \triangle t$$

*Jacobi method (*JAC *method):* Instead of discretizing the CTMC one employs here a decomposition. The generator matrix Q can be partitioned into a matrix of its diagonal elements and the rate matrix R, yielding $Q := R - D$. One may note that matrix D carries the exit rates of the respective states, whereas matrix Q carries the negative exit rates on its diagonal. The iteration scheme of the *JAC* method is then given as follows:

$$\vec{\pi}^{k+1} = \vec{\pi}^{k} \cdot RD^{-1}$$

which can be written in an element-wise notation as follows:

$$\pi_i^{k+1} := \frac{1}{|q_{i,i}|} \sum_{\substack{j \in \mathbb{S} \\ i \neq j}} \pi_j^k Q(j,i)$$

*Gauss-Seidel method (*GS *method):* Similar to the *JAC* the *GS* method also employs matrix decomposition: $Q := D - T$, where D carries now the negative exit rates of the respective state, i. e. $d_{i,j} := q_{i,j}$ for $i = j$ and 0 otherwise. The transition rate matrix T can be decomposed further $T := (L + U)$, where L and U are the negative lower and upper triangular portions of the transition rate matrix T. The iteration scheme is then given for two variants:

(1) Forward *GS* method: $\vec{\pi}^{k+1} = \vec{\pi}^{k} \cdot L(D - U)^{-1}$. This can be written in an element-wise notation as follows:

$$\pi_i^{k+1} := \frac{1}{|q_{i,i}|} \left(\sum_{\substack{j \in \mathbb{S} \\ j < i}} \pi_j^{k+1} Q(j,i) + \sum_{\substack{j \in \mathbb{S} \\ j > i}} \pi_j^k Q(j,i) \right) \tag{2.5}$$

(2) Backward *GS* method: $\vec{\pi}^{k+1} = \vec{\pi}^{k} \cdot U(D - L)^{-1}$. The element-wise notation is achieved by simply swap $\vec{\pi}^{k+1}$ and $\vec{\pi}^{k}$ on the right side of the Eq. 2.5.

In contrast to the *JAC*, the *GS* method already employs the newly computed state probabilities for all states having a lower index as the state of the current row ($j < i$), for states having a larger index one employs the probabilities of the last iteration –in case of the backward method this is simply the other way round. This implies that the *GS* method has a better convergence, but this also requires an ordered access (row- or column-wise access) to the state probabilities and thus matrix elements of Q, at least for the states the new probability of which is needed.

Over-relaxation For improving the convergence of the *JAC* and *GS* method, one may build the weighted sum of the state probabilities of the previous and of the recently finished iteration for obtaining the state probabilities employed in the next iteration. This is commonly denoted as over-relaxation. Together with the above introduced iteration schemes this yields the Jacobi over-relaxation method (*JOR*) and the forward or backward *GS* over-relaxation method (*GSOR*). –However, choosing the weight for computing $\tilde{\pi}_i^{k+1} := (1-\omega)\tilde{\pi}_i^k + \omega\pi_i^{k+1}$, turns out to be difficult in practice, even though ω is known to be restricted to the interval $(0, 2)$

Computing rewards
For simplicity we defined rate and impulse rewards as being state-/activity-dependent functions. However, the underlying stochastic process is a CTMC, consequently as the system evolves over time, so do the reward values. Thus it is required that we extend the time-independent reward functions defined in Def.2.3 and 2.4 with a notion of time. However, it is essential to note that the reward value induced by a specific state or transition remains time-independent. We will now roughly investigate the concept of rate and impulse rewards. The interested reader may refer to [San88, MS91, CBC+93, Ger00] for further details.

Rate reward
A rate reward is the cost or gain obtained while being in a state i. Thus the rate reward obtained in a specific state i at time point t can be computed as follows:

$$\mathcal{R}_r(i,t) = \pi_i(t) \cdot \mathcal{R}_r(i) \tag{2.6}$$

where $\pi_i(t)$ is the probability for being in state i at time point t and $\mathcal{R}_r(i)$ is the time-independent rate reward value of state i concerning rate reward r (cf. Def. 2.3). The probability $\pi_i(t)$ employed in the above equation can be computed by executing the methods illustrated above, where for arbitrary time points t a transient and for $t \to \infty$ a steady state analysis is applicable. Since each state $i \in \mathbb{S}$ has its own rate reward value with respect to reward function r, one must simply sum the reward values over all states yielding the state-independent reward value $\mathcal{R}_r(t)$ at time-point t:

$$\mathcal{R}_r(t) := \sum_{i \in \mathbb{S}} \mathcal{R}_r(i,t) \tag{2.7}$$

So far we only computed instant-of-time rewards, however also interval-of-time and time averaged interval-of-time rewards are from concern. A rate reward obtained for a time interval $[t, t+\triangle t]$ can be compute as follows:

$$\mathcal{R}_r(t, t+\triangle t) := \sum_{i \in \mathbb{S}} \mathcal{R}_r(i) \cdot \bar{\pi}_i(t, t+\triangle t) \cdot \triangle t \tag{2.8}$$

where $\bar{\pi}_i(t, t+\triangle t)$ is the average state probability for being in state i during time interval $[t, t+\triangle t]$. It can be computed as follows:

$$\bar{\pi}_i(t, t+\triangle t) := \frac{1}{\triangle t} \int_{\tau=t}^{t+\triangle t} \pi_i(\tau)\delta\tau$$

By norming the computed value to the time period analyzed ($\frac{1}{\triangle t}$), the above interval-of-time reward measure can converted into time-averaged value. Now one may investigate the behavior of the interval-of-time rewards, if $t \to \infty$ or $t + \triangle t \to \infty$. So in case of steady-state $\pi_i(t, t+\triangle t)$ needs solely to be replaced with the steady-state distribution π_i, where in case of a time-averaged reward a subsequent norming to the length of the time interval must follow. This should suffice for the time being, since a deeper discussion is out of scope of this work.

Before we proceed one may also note that due to the above definition of rate rewards, only those states contribute to $\mathcal{R}_r$ which have a state probability different from 0. We will come back to this issue in the next section, where SG reduction techniques are covered.

Impulse reward

An impulse associated with a specific transition is obtained, each time the respective transition is taken by the system, i.e. a transition $i \xrightarrow{\lambda_{i,j}} j$ may contribute to the overall value of an impulse reward $\mathcal{I}^a$. One may compute then the impulse obtained during the time interval $[t, t+\triangle t]$ by a single transition as follows:

$$\mathcal{I}^a(i,j,t,t+\triangle t) := \bar{\pi}_i(t,t+\triangle t) \cdot \triangle t \cdot \mathcal{I}^a(i,j) \cdot \lambda_{i,j} \tag{2.9}$$

where $\bar{\pi}_i(t,t+\triangle t)$ is the average state probability for being in state i during time interval $[t,t+\triangle t]$ as already employed before.

Since there might be more than one transition emanating form state i and contributing to impulse reward $\mathcal{I}^a$ it follows:

$$\mathcal{I}^a(i,t,t+\triangle t) := \sum_{j\in\mathbb{S}} \mathcal{I}^a(i,j,t,t+\triangle t) = \pi_i(t,t+\triangle t) \cdot \triangle t \cdot \sum_{j\in\mathbb{S}} \mathcal{I}^a(i,j) \cdot \lambda_{i,j} \tag{2.10}$$

In order to obtain the "state-independent" impulse reward $\mathcal{I}^j(t)$ one simply needs to sum over all states, yielding:

$$\mathcal{I}^a(t,t+\triangle t) := \sum_{i\in\mathbb{S}} \mathcal{I}^a(i,t,t+\triangle t) = \sum_{i\in\mathbb{S}} \sum_{j\in\mathbb{S}} \mathcal{I}^a(i,j,t,t+\triangle t) \tag{2.11}$$

So far we only computed an interval-of-time impulse reward. By norming the computed values to the length of the time-interval ($\triangle t$), the above interval-of-time reward measures can be converted into time-averaged values.

In case of steady-state we restrict the discussion to time-averaged impulse rewards, so that $\pi_i(t,t+\triangle t)$ in Eq. 2.9 can be replaced with the steady-state distribution π_i, where a subsequent norming to the length of the time interval of interest must follow. This yields:

$$\tilde{\mathcal{I}}^a(i,j) := \pi_i \cdot \mathcal{I}^a(i,j) \cdot \lambda_{i,j} \tag{2.12}$$

If this is employed in Eq. 2.10 and 2.11 one obtains:

$$\tilde{\mathcal{I}}^a(i) := \sum_{j\in\mathbb{S}} \tilde{\mathcal{I}}^a(i,j) = \pi_i \cdot \sum_{j\in\mathbb{S}} \mathcal{I}^a(i,j) \cdot \lambda_{i,j} \quad \text{and} \quad \tilde{\mathcal{I}}^a := \sum_{i\in\mathbb{S}} \sum_{j\in\mathbb{S}} \tilde{\mathcal{I}}^a(i,j) \tag{2.13}$$

which is the average impulse reward (value) obtained in steady-state for impulse reward a.

2.2.3 Reduction techniques

The number of states a MRM consists of is a bottleneck for the SG based analysis, since as pointed out above, individual state probabilities must be computed for evaluating rate and impulse rewards. Therefore an important goal can be seen in the reduction of the number of states. This reduction can either be done on-the-fly, i.e. while transforming the high-level model description into a low-level MRM, or a posterior to state space exploration.

Elimination of vanishing states

Due to the nature of the high-level model description methods, as it will be discussed in Sec. 2.3, it appears that after transformation of the high-level model into its low-level representation two kinds of transitions between pair of states exist, immediate and timed ones. Timed transitions are taken with an exponential delay, whereas instantaneous transitions are taken immediately. It is is evident that within states with outgoing immediate and timed transitions, the latter will never be taken ("win the race"). This yields that the system will spend only time in states which can exclusively be left via outgoing timed transitions. States of such kind are denoted as tangible, whereas states to be left via immediate transitions are

denoted as vanishing. –States which can not be left are denoted as absorbing states, (see discussion above).– In case a vanishing state can be left via more than one immediate activity, the non-determinism has to be resolved. This is commonly done by assigning probabilities to each immediate transition, where the probabilities of all immediate transitions emanating from the same state must sum up to 1. As results one yields a transition matrix T, where some entries refer to transition probabilities and some entries refer to transition rates. As known from the literature, e.g. [BCD+95, CBC+93], T can now be converted into a pure transition rate matrix, defining a proper CTMC, which is achieved by eliminating all entries referring to vanishing states. Doing so is justified, since a vanishing state does not contribute to the overall rate reward $\mathcal{R}_r$ as π_i is zero. In case of a impulse reward the situation is more complicated. For simplicity we solely allow timed transitions to induce impulse rewards, so that vanishing states and their outgoing transitions are also irrelevant here. –For a discussion on impulse rewards induced by immediate transitions the reader may refer to [Ger00].

The construction of the proper CTMC from a transition matrix of the above kind, can be achieved by the following steps:

(1) Partitioning of the transition matrix T :

$$T = \begin{pmatrix} T_{t,t} & T_{t,v} \\ P_{v,t} & P_{v,v} \end{pmatrix} \tag{2.14}$$

where $T_{t,t}$ and $T_{t,v}$ are the sub-matrices which solely contain transition rates, so that $T_{t,t}$ refers to the transition rates among the tangible states and $T_{t,v}$ refers to the transition rates from tangible to vanishing states. Matrix $P_{v,t}$ and $P_{v,v}$ are the sub-matrices, which contain the transition probabilities, in case of $P_{v,v}$ between the vanishing states and in case of $P_{v,t}$ from the vanishing to the tangible ones.

(2) Calculating the proper transition rate matrix T' :
Now one is able to calculate a transition rate matrix which represents the effective transition rates between the tangible states, after the vanishing states have been eliminated.

$$T' = T_{t,t} + T_{t,v} N P_{v,t} \tag{2.15}$$

where $N = \sum_{n=0}^{\infty} P_{v,v}^n = (1 - P_{v,v})^{-1}$. This approach reduces the SG, because the vanishing states are eliminated and the $(n \times n)$ transition matrix T is reduced to the $(n_t \times n_t)$ transition *rate* matrix T'.

The disadvantage of the above scheme under traditional SG exploration techniques is quite clear, it requires a full expansion of all reachable states, before reduction can be done. Therefore many tools make use of an on-the-fly reduction method, where vanishing states are either stored temporarily, or they are not stored at all. However, as it will be discussed in this thesis, for symbolic methods the generation and storage of large SGs is is not an issue. Thus it seems useful to execute the elimination of vanishing states a-posteriori to SG generation, especially since this simplifies the handling of time-less traps.[1] The respective symbolic algorithms can be found in [Sie02].

Lumping the states of a MRM

The idea of state lumping [San88, Buc94, Sie95, BCD+95] is as follows: One partitions the state space of a CTMC and lump all states belonging to the same partition. The states of a partition are then represented by a dedicated representative, often also denoted as macro state. The transition rates among the resulting macro states need hereby to be adapted accordingly, so that one solely needs to compute the probability distribution on the set of partitions, rather than on the set of individual system states. Since the number of partitions

[1] A BSCC containing vanishing states only, is commonly denoted as time-less trap.

is in general much smaller than the number of states the advantage of such a procedure is obvious.

Different levels of lumpability exist, which we define as follows:

Definition 2.6: Lumpability of states

Let $C := (\mathbb{S}, T, \vec{\pi}(0))$ be a CTMC and $P := \{\mathbb{S}_1, \mathbb{S}_2, \ldots \mathbb{S}_n\}$ a partitioning of $\mathbb{S}$.

(1) If for $k, l \in \{1, \ldots, n\}$ and $\forall i, j \in \mathbb{S}_k$:

$$\sum_{x \in \mathbb{S}_l} T(i, x) := \sum_{x \in \mathbb{S}_l} T(j, x)$$

holds, one denotes C as ordinarily lumpable.

(2) If for $k, l \in \{1, \ldots, n\}$ and $\forall i, j \in \mathbb{S}_k$:

$$\pi_i(0) = \pi_j(0) \text{ and}$$
$$\sum_{x \in \mathbb{S}_l} T(x, i) := \sum_{x \in \mathbb{S}_l} T(x, j)$$

holds, one denotes C as exactly lumpable.

(3) If for $k \in \{1, \ldots, n\}$ and $\forall i, j \in \mathbb{S}_k$ the conditions of ordinary and exact lumpability are satisfied, one denotes C as strictly lumpable.

One may note that only in case of exactly lumpable states, the individual state probabilities of the states of a partition are obtainable ($\frac{\pi_{\mathbb{S}_k}}{|\mathbb{S}_k|}$), in case of ordinary lumpability these probabilities are not available.

What left out so far, is the lumpability of states annotated with reward values, i.e. so far we were only concerned with the states of a CTMC. Ordinary lumpability turns out to be problematic, since as pointed out above, the individual state probabilities can here not be derived from the probability distribution on the set of macro states. Therefore it is essential to augment the ordinary lumpability defined above with the requirement that two states of a MRM $M := (C, \mathcal{I}, \mathcal{R})$ are only ordinarily lumpable if their rate reward values or the impulse reward values of their emanating transitions are identical, i.e. for $k, l \in \{1, \ldots, n\}$ and $\forall i, j \in \mathbb{S}_k$ the following condition must be satisfied:

$$\forall \mathcal{R}_r \in \mathcal{R} : \mathcal{R}_r(i) = \mathcal{R}_r(j) \text{ and}$$
$$\forall \mathcal{I}^a \in \mathcal{I} : \sum_{x \in \mathbb{S}_l} \mathcal{I}^a(i, x) T(i, x) = \sum_{x \in \mathbb{S}_l} \mathcal{I}^a(j, x) T(j, x)$$

One may note that a partitioning satisfying the criteria of exact lumpability defines an equivalence relation on the set of states, thus one also speaks of class of states rather than partitions. The equivalence relation on MRMs, the transitions of which are additionally equipped with labels, is commonly denoted as Markovian bisimulation. I.e. weak or strong Markovian bisimulation can be seen as refinement of lumpability, since contrary to Markov chains, bisimulation considers transition labels, so that not only the aggregated rates must match but also the labels of the transitions. Details can be found in [Göt94, Her98]. For simplicity we will speak of bisimulation only, but in fact are referring to (strong) Markovian bisimulation.

The next question to be dealt with, is to answer how the transition rates among the macro states need to be adjusted if (a) the partitioning of the state space is achieved (b) strict lumpability is satisfied and (c) one wishes to construct a reduced MRM having the same solution concerning the performability measures as the original, un-reduced MRM. We will an-

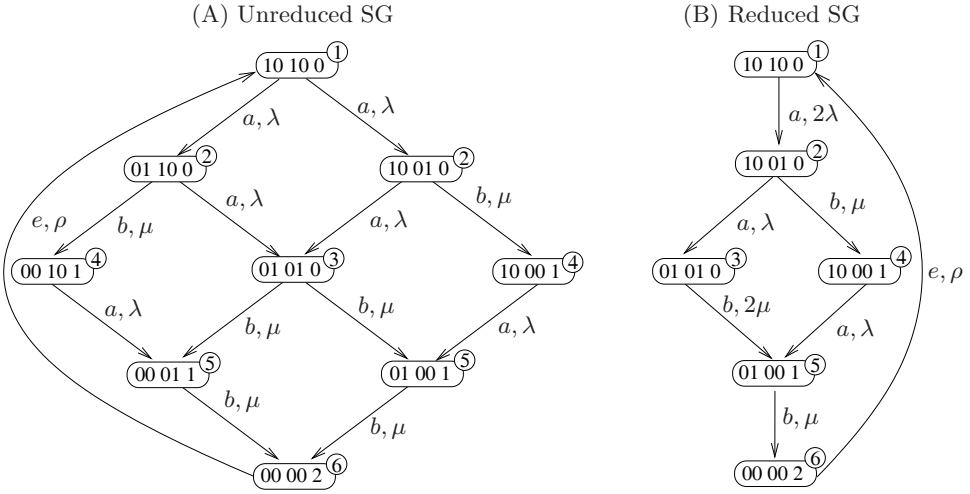


Figure 2.1: Bisimilar SGs

swer this question informally, further details can be found in [San88, Buc94, Sie95, BCD+95].

For constructing a reduced MRM from its un-reduced counterpart one may proceed as follows:

(1) Mark each state according to its class $\mathbb{S}_i$.

(2) For each class $\mathbb{S}_i$ pick the class representative r_i. For $i, k \in \{1, \ldots, n\}$ and $i \neq k$ the (aggregated) rate $\tilde{\lambda}_{r_i,r_k}$ of the transition connecting state r_i with state r_k within the reduced MRM can be computed by $\tilde{\lambda}_{r_i,r_k} := \sum_{x \in \mathbb{S}_k} \lambda_{i,x}$.

For exemplification one may finally refer to Fig. 2.1.A, where a MRM, the transitions of which are equipped with rates and labels, is illustrated. Let the initial probability distribution on states be uniformingly distributed. Consequently this MRM fulfills the conditions of strict lumpability, where we numbered the equivalence class and labeled the states accordingly. –In fact, the partitioning not only fulfills exact lumpability but also (strong) Bisimulation, since also the labels of the transitions connecting the states of the different equivalence classes match.– In the next step, we pick a class representative for each class, and compute the aggregated or cumulative rates according to the above procedure. E.g. state (10 10 0) reaches two states of class 2, consequently the cumulative rate can be evaluated to 2λ. Contrary to this, the representative of class 2, the state (10 01 0), is solely connected to one state of class 3 and one state of class 4, thus no transitions need to be aggregated here. The completely lumped SG is illustrated in Fig. 2.1.B.

The major problem of state lumping is the finding of a partitioning P, so that the criteria of Def. 2.6 are matched. As simplification [San88, Sie95] suggest the construction of high-level models in a structured manner, so that the partitions of the state space to be lumped are known, But this procedure may not necessarily yield the minimal equivalence relation on the set of states. We will comeback to this in Sec. 2.4 and 2.5.

2.2.4 State/Transition systems

For representing finite Markov Reward models, this thesis employs various types of state-transition systems (ST systems), which will be introduced in the following. The discussion follows hereby an order of inheritance, so that the characteristics required for the more simpler forms of transitions are also assumed to be decisive for the more complex ones.

A ST system consists of a (finite) set of states ($\mathbb{S}$), and a transition function. A transition function is a mapping $\Delta : \mathbb{S} \mapsto \mathbb{S}$, yielding a predecessor/successor relation $\rightarrow \subseteq \mathbb{S} \times \mathbb{S}$ on the set of states. If each directed edge, connecting a predecessor with its successor state,

is labeled with a symbol $l \in \mathcal{A}ct$ one speaks of a labeled transition system (*LTS*), yielding the relation $\rightarrow \subseteq \mathbb{S} \times \mathcal{A}ct \times \mathbb{S}$. In the context of this work we require that all edges of a set of edges, connecting the same pair of successor and predecessor state (i, j), carry different labels. Besides labels, transitions can also be equipped with probabilities $p \in [0, 1]$ or rates $r \in \mathbb{R}_0^+$. A *LTS*, where transitions are exclusively equipped with probabilities, is denoted probabilistic *LTS* (*pLTS*), so that $\rightarrow \subseteq \mathbb{S} \times \mathcal{A}ct \times [0, 1] \times \mathbb{S}$. The probabilities of all edges emanating from the same state i are required to sum up to 1. In case all transitions are equipped with rates only, one speaks of a stochastic *LTS* (*sLTS*), where $\rightarrow \subseteq \mathbb{S} \times \mathcal{A}ct \times \mathbb{R}^+ \times \mathbb{S}$. Transition systems where the directed edges are either labeled with a probability or a rate are denoted as extended *sLTS* (*esLTS*). For *esLTS* it is required that all edges emanating from a state i are either all probabilistic, i.e. they carry a probability or that they are stochastic, i.e. they carry a rate. The source states of probabilistic transitions are denoted as vanishing states, whereas the source states of stochastic transitions are denote as tangible states. Analogously to a *pLTS* it is also required that the probabilities of the transitions emanating from the same (vanishing) state sum up to 1. Given an *esLTS*, *sLTS* or *pLTS* one obtains the respective non-labeled variant by removing the labels from the directed edges, where edges connecting the same pair of states are merged. The resulting edge is hereby equipped with the cumulative rate or probability, which is the sum of the individually edges to be merged [Göt94].

Form a *sLTS* S a CTMC $C := (\mathbb{S}, T, \vec{\pi}(0))$ can be constructed as follows:

(1) The states of S' are the states of the CTMC.

(2) Remove all labels and make S a non-labeled *sTS* S', according to the above procedure.

(3) The transition rate matrix T is constructed as follows:

$$T(i,j) = \begin{cases} \mu \;\Leftrightarrow (i, a, \mu, j\,) \in S \\ \qquad \wedge\, i \neq j \\ 0 \qquad \text{else} \end{cases} \tag{2.16}$$

(4) The initial probability distribution is defined as follows:

(4.a) $\forall i \in \mathbb{S} : \pi_i(0) := 1/|\mathbb{S}|$ or

(4.b) for one $i \in \mathbb{S} : \pi_i(0) := 1$ and $\forall j \in \mathbb{S}$ where $j \neq i : \pi_j(0) := 0$.

Thus in the context of this thesis each *sLTS* is directly interpreted as a CTMC. Due to the nature of the high-level model description methods, it appears that after transformation of the high-level model, one obtains an *esLTS*, rather than a *sLTS*. But as illustrated in Sec. 2.2.3 as far as it is from concern for this work, the transitions equipped with probabilities and the states they emanate from can safely be eliminated, so that one ends up with a proper *sLTS* and thus a proper CTMC. In the following paragraphs we will therefore generically speak of SGs, in order to refer to a *sLTS* or *esLTS*, in case the difference is significant the precise notation will be employed.

2.3 High-level Markov reward models

The introduction to high-level stochastic modeling formalisms will be followed by briefly indicating how performability measures on the level of high-level models can be defined. Compositionality plays an important role, since it follows the design principle of modularity. Thus it avoids the error-prone specification of monolithic models, which, as matter of size, might tend to become unreadable. Therefore we will also cover methods for constructing high-level models from smaller components in this section. How high-level models can be mapped to MRM via SG generation is briefly explained at the end of this section, a profound discussion, as far as it is from concern for this thesis, will follow in the next sections.

2.3.1 High-level model description techniques

Over the past years several powerful methods as known from the functional analysis of systems, have been extended to the Markovian case. In the following we will briefly introduce some of them. However, a deeper discussion is avoided, since a large amount of introductory and advanced literature for this topic exists. Before we proceed it is essential to note that all high-level model specification methods have in common, that each model S consists of a finite ordered set of discrete state variables (SVs) $\mathfrak{s}_i \in \mathfrak{S}$, and a finite set of activities ($\mathcal{A}ct$). In contrast to the standard Petri net nomenclature, we speak of activities when referring to the high-level constructs. Thus transitions in the context of this work are the connection between system states, established each time an activity has been executed. This differentiation si useful, as far as high-level model description methods are from concern, where system states are not part of the specification method, e.g. Petri nets, process algebras, etc.. In case "simpler" modeling methods, where system states are part of the description technique, we will also speak of transitions.

Stochastic automata networks

Stochastic automata networks [Pla85, Buc91, Ste94] are a very low-level modeling approach, since high-level model description and its low-level representative are almost identical. A stochastic automata consists of states and activity-labeled transitions among them, where the latter may also be equipped with a stochastic rate or they may not. In the former case one specifies an exponentially delayed transition, whereas in the latter case an un-delayed behavior is modeled. In consequence a stochastic automata can be viewed as a Markov model with tangible and vanishing states. In order to compactly specify complex systems, the modeler is enabled to combine sets of stochastic automata by activity synchronization (cf. Sec. 2.3.3). This gives one a network of stochastic automata, naturally describing the behavior of a system in a compositional way. Since the individual stochastic automata do not contain any local variables the state of a stochastic automata network is naturally described by a set of local state counters, each referring to the state of a specific stochastic automata.

Stochastic state charts

In recent years, state charts like the ones employed in the modeling standards SDL or UML, have been extended to the Markovian case. In its simplest form a stochastic state chart may consist of a set of states, variables and transitions among these states, which may modify the variables when executed. Thus a state of the state chart can be described by the current values of the local variables and an additional state counter. In order to specify timed behavior one may think now of equipping transitions with stochastic rates, referring to an exponential delay of the associated transition's execution. Transitions of this kind are commonly denoted as Markovian transitions. In order to allow also the employment of case distinctions, one may also make use of a special node, its incoming edge is a Markovian transition, and its outgoing edges are equipped with execution probabilities. Since the outgoing edges are taken with a zero delay, they are commonly denoted as immediate or instantaneous transitions. By introducing the concept of initial and terminal states and referencing of (sub-) state charts, state charts can be organized in a modular fashion. However, it seems to be straight-forward, to also allow the composition of state charts via the sharing of variables and/or the joint execution of activities (cf. Sec. 2.3.3). A sophisticated extension of UML state charts can be found in [JHK03].

Generalized stochastic Petri Nets

A generalized stochastic Petri Net (GSPN) is a bi-partite graph, which consists of a set of activities and a set of places. Each place may contain an arbitrary number of tokens. Here a state of the system can be described by the current number of tokens contained in each place, so that each place of the GSPN gives a SV of the overall model. Activities and places are connected via the definition of a connection relation. The execution behavior of activities

is then steered by its connected input places, whereas its execution will manipulate the contents of the places contained in the activity's set of output places. For specifying timed behavior, activities are either executed after an exponential delay or instantaneously, where the race condition among the competing activities must be resolved. A profound overview over GSPNs can be found in [BCD+95].
In order to organize a GSPN in a modular way, so that its readability is improved, one may combine different(sub-)nets via the sharing of places. Furthermore concepts known from stochastic automata and stochastic process algebras have been extended to the case of GSPNs, so that also the composition via activity-synchronization can be found, e.g. [CT96, HHMR97] (cf. Sec. 2.3.3).

Stochastic Activity Networks
Stochastic Activity Networks (SAN), introduced in [San88], are an extension of GSPNs. Consequently a state of a SAN can also be described as a tuple of SVs, where each refers to a specific place of the net. In addition to GSPNs, SANs allow the use of so called input and output gates. These gates can be seen as the enrichment of the enabling predicates (guards) and the execution functions of the connected activities. SANs also allow the association of each activity with a set of cases, where each case needs to be parameterized with an execution weight. Simply speaking a case is the execution of an activity, subsequently followed by an execution of a set of instantaneously activities, where the individual execution probability is determined by the specific case-individual weight. In contrast to GSPNs, SANs allow the use of other than exponential distributions for describing the timed delay of activity executions. However, in the context of this work, we are occupied with Markov models only, thus we only allow activities to have exponential or un-timed behavior. Analogous to GSPNs, SANs can be combined via the sharing of places or as realized recently via the joint execution of activities.

Stochastic process algebras
Within a stochastic process algebra (SPA) a process specification consists of operators for choices, guarded choices, process variables, individually labeled activities, and most importantly of a reference to the subsequent behavior at process termination, e.g. stop, or calling another process. Consequently the state of the process may be described by the values of the local process variables and a process counter. Activities can be timed, i.e. they are equipped with rates, or they are instantaneous. Examples of stochastic process algebras can be found in [Göt94, Hil94a, HHK+98, HHM98].

A very important concept, which has been developed in the context of PA, is constructivity of the formalism: (a) Like stochastic automata a system can be built in a compositional manner, where activity-synchronization (cf. Sec. 2.3.3) is employed for combining the individual process instances. (b) Another feature of SPAs is the possibility of hiding internal behavior (activities) from the environment, which may lead via exploitation of bisimulation to fewer system states and thus simplify the SG based analysis. (c) A very strong concept is the axiomatization of SPAs, and the notion of equivalence of processes. This allows one to replace processes with simpler ones, so that the overall system is much smaller, but exhibits the same functional and timed behavior. In recent years, aspects of constructivity have been adopted to other high-level model composition methods, where especially the compositional construction of high-level models plays an important role [HHMR97]. Since most SPAs also allow to specify local variables, a composition of submodels via the sharing of variables seems applicable, but may interferes with the above explained concept of constructivity.

2.3.2 Specification of performability measures
The elementary units of performability measures are reward values, associated with each state or transition of a MRM underlying a high-level model description. However, for maintaining transparency, most modeling tools enable the user to specify reward functions on

the level of the high-level model description.
In this context a rate reward r is a function, which takes a subset of the SVs of the high-level model as its function variables. Depending on the values of these SVs a specific rate reward value can be computed. In contrast to rate rewards, impulse rewards are associated with the completion of activities. Therefore the user simply defines a reward value obtained if a specific activity is executed. However, this reward value may also depend on the values of some SVs, which makes each impulse reward activity and SV-dependent. A set of rate and impulse reward functions defined on the high-level model can than be aggregated, in order to establish complex performance variables (PV), where the value of a PV p is the sum of a set of rate or a sum of a set of impulse reward functions or the sum of both.

2.3.3 Composition of high-level model descriptions

A high-level model can be constructed in a modular and hierarchic way by specifying a set of submodels and define the way of how they are interacting.

Pure interleaving

If no interaction among the submodels takes place one speaks of pure interleaving, i.e. the submodels, which are in fact (disjoint) partitions of the overall model are executed concurrently. Pure interleaving is the special case of composition via joint activity execution and composition via the sharing of SVs, as discussed next. There pure interleaving is obtained, if the sets of objects the composition is archived by is empty.

Joint activity execution (*Sync*)

When composing submodels via joint activity execution, the submodels are executed in parallel, where a subset of dedicated activities has to be executed jointly by all participating partners. Different approaches concerning the type of synchronizing activities exists, where also the computation of the rate of the synchronized activities is the target of discussion. In the following we will employ the operator $S_1 \|_{\mathcal{S}} S_2$ for specifying the synchronization of submodel S_1 and S_2 over all activities appearing on the set $\mathcal{S}$, but where solely activities with the same label are executed synchronously.

Sharing of SVs (*Join*)

If a high-level model description method employs (local) variables, it is possible to compose submodels, specified in the respective formalism, by merging sets of local variables ($\mathcal{J}$) [San88]. This technique is commonly denoted as sharing or joining of SVs (*Join* for short). We we will employ the operator $S_1 <\mathcal{J}> S_2$ for specifying that submodel S_1 and S_2 share the variables appearing on the set $\mathcal{S}$, but where solely SVs with identical labels are merged.

Replication of submodels (*Rep*)

A *Rep* causes the multiple instantiation of a dedicated submodel. These instances can then expose information for further composition to the environment, where one may differ between the sharing of variables or the joint execution of activities. In [San88] the sharing of a dedicated variable with the environment is introduced, where the respective variable is also shared among all instances of the replicated submodel (*Rep**Join*). A synchronization of activities is the strategy introduced in [Sie95], where all instances of the replicated submodel synchronize on the same set of activities (*Rep**Sync*).
It is important to note, that the replication of dedicated submodels induces strong bisimularity on the level of a high-level model's underlying (activity-labeled) MRM. In the following we will denote this bisimulation as submodel-imposed symmetry.

2.3.4 Mapping of high-level models to MRMs

According to the above discussion, one can assume that each model M consists of a finite ordered set of discrete state variables (SVs) $\mathfrak{s}_i \in \mathfrak{S}$, and a finite set of activities ($\mathcal{A}ct$), which are either of Markovian ($\mathcal{A}ct^m$) or instantaneous nature ($\mathcal{A}ct^i$). Concerning the high-level model description by means of GSPNs, SANs or SPAs, each $\mathfrak{s}_i$ records hereby the number of tokens in a place, the state of the program counter, the values of the process parameters, etc.. Since the set of SVs is finite and ordered, each state of the model can be written as a vector $\vec{s}$. The i'th component of a (state) vector $\vec{s}$, addressed by the notation $\vec{s}[i]$, yields the current value of SV $\mathfrak{s}_i$ in state $\vec{s}$. During SG generation, as already briefly introduced in Sec. 1.2, on successively visits all possible system states and generates all possible transitions among them. This procedure allows one to map a high-level model to an *esLTS*, which can be interpreted as a CTMC once the vanishing states have been eliminated. In order to equip the CTMC with rate and impulse rewards, one may also execute the rate and impulse reward functions during SG generation, so that one yields a reward value for each state or transition. This allows one then to interpret the generated and reward annotated SG as (low-level) MRM, its solution (cf. Sec 2.2.2, p. 11ff) will given one the desired performability measures as specified by the modeler on the high-level model description. In the next sections we will briefly explain, how the SG generation can be organized. Two classes of strategies can be recognized: (a) the non-compositional or monolithic procedures, which executed the SG generation for the complete model and (b) the compositional SG generation procedures, where the SGs of the overall models are constructed from smaller components, commonly from the SGs of user-defined submodels, and thus representation of SGs can be restricted to the submodels, given that a respective composition scheme is provided.

2.4 Non-compositional state graph construction

If the SG of a high-level model is not constructed from smaller components, one commonly speaks of a non-composition or monolithic SG generation procedure. For handling the composition operators, so that one is enabled to also carry out a monolithic SG construction procedure for compositionally constructed high-level models, additional effort is required:

(1) *Sync*: For resolving the *Sync*-composition operator on the level of the high-level model description, one simply needs to build the *conjunction* of the entrance or enabling conditions and of the execution functions of all activities to be executed synchronously. Since the SG generation is monolithic, activity priorities, rates and weights of the newly obtained activity can be chosen arbitrarily [Lam00].

(2) *Join*: For sharing SVs among submodels, one simply needs to map the local SVs to be shared to the same new SV, so that the submodels are enabled to invisibly, mutually manipulate their behavior [San88].

(3) *Rep*: For realizing the *Rep*-operator one simply allocates multiple instances of the respective submodel. Their merging is than achieved by either executing a *Sync* or a *Join* on the level of the resulting high-level model description. As it will be pointed out below, the *Rep*-operator can already be exploited by a monolithic SG generation procedure, so that one is enabled to generate a reduced (bisimilar) SG on-the-fly.

After applying the above transformations, one is enabled to carry out a monolithic SG generation procedure, even for compositionally constructed models. For constructing the SG of the overall model, one executes each enabled activity of the high-level model, one at a time, for each visited state, so that the model evolves from one state to another, where each SV $\mathfrak{s}_i$ can take an arbitrary value from $\mathbb{N}$. This procedure yields the set of reachable states $(\mathbb{S})$[2] and the set of transitions among them, where the elements of $\mathbb{S}$ are mappings of the following kind:

[2] Reachable means, that there exists a sequence of activity executions which brought the state about, starting in the initial state (this will be formalized in Sec. 4.2.2, Def. 4.9, p. 77).

$$\mathcal{M} : (\mathfrak{s}_1, \ldots, \mathfrak{s}_{|\mathfrak{S}|}) \rightarrow \mathbb{N}^{|\mathfrak{S}|}, \tag{2.17}$$

and $\mathbb{S} \subset \mathbb{N}^{|\mathfrak{S}|}$. As long as the high-level model's underlying SG is finite the above procedure will terminate.

Exploiting user-defined symmetries

As already mentioned previously, the employment of a *Rep*-operator yields strict lumpability of states, or to be precise strong Markovian bisimularity. This can be exploited for generating a reduced SG on-the-fly as follows:

For constructing a reduced SG on-the-fly each visited state is immediately replaced by the macro state of the partition it belongs to, which yields the following effects: (a) One only explores one state per partition, namely the macro state. (b) When generating the successor states of a (macro) state one executes all enabled activities, where the target states are once again immediately replaced by the partition representatives, and transitions connecting the same pair of states are aggregated. I.e. one obtains a single transition between pair of states, which is equipped with the cumulative rate. The main problem is now to determine all states which belong to the same partition, so that each state can be replaced by the state descriptor of its macro state. However, in the presence of explicitly modeled submodel-imposed symmetries, this is no problem, the partitions and thus macro states of each visited state can be determined on-the-fly. For simplification one may consider now a high-level model consisting of m distinct and k identical submodels, and that states of identical submodels are labeled in exactly the same way. Thus a state of the overall model is given as a vector containing $(m + k)$ sub-states, where each of the latter refers to a specific unique or replicated submodel. A partition of the overall model contains now all states, where the state vector is a permutation of the k sub-states, but carries identical values on the m remaining positions. This allows one to transformed the k sub-states into a canonical representation [San88], or to order the k sub-states in such a way that always the lexicographic largest or smallest state descriptor is produced [Sie95]. This allows an easy mapping of each newly reached state to its macro state.

One may note that even the above procedure exploits the compositional structure of high-level models, the SG construction procedure is still monolithic, since the overall model's SG is not constructed from smaller components.

2.5 Compositional state graph construction

When generating the SG of a high-level model, the compositional structure of the latter can be exploited. I.e. it might be possible to construct the SG of the overall model from smaller components, e.g. the SGs of the submodels. This, which we denote as compositional SG construction, may yield the following advantages:

(1) The runtime of the SG generation procedure may be reduced, since not all sequences of independent activities have to be extracted explicitly. –This of course holds only for symbolic compositional techniques, where states can be computed, rather than explicitly generated.

(2) It is known that in case of activity-synchronization and for a limited class of *Join*-composed models the transition rate matrix of the overall model can be represented by a Kronecker operator based expression of submodel-local transition rate matrices, so that the storage of the overall transition rate matrix is unnecessary.

(3) One may replace local SGs by smaller ones, as long as the information required for composition is preserved, which is commonly denoted as compositional SG reduction.

Consequently approaches exploiting compositionality on the level of SGs are possibly not only more run-time-efficient but also more memory-efficient.

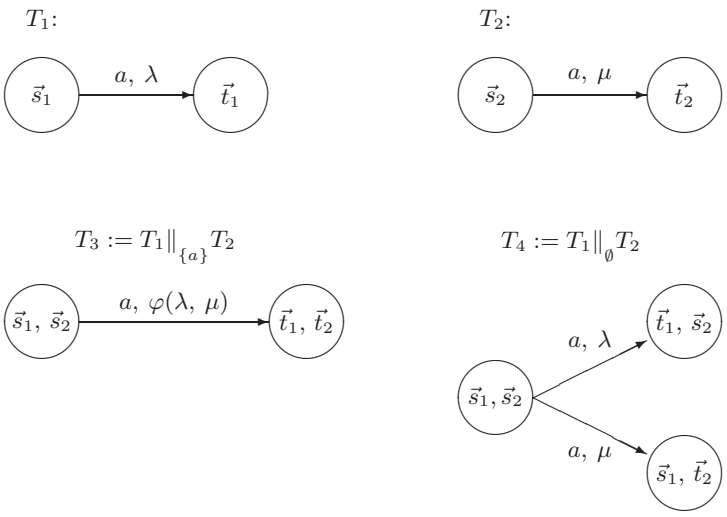


Figure 2.2: Compositional SG construction for interleaving and synchronization

In the following paragraphs we will briefly explain the semantics of high-level model composition techniques on the level of SGs. Hereby the Kronecker operator (*KO*) driven composition of local SGs [Pla85, Buc91, Sie95] for constructing an overall model's CTMC will be discussed along the way. The definition of the respective operators is given in appendix A.3. The compositional construction of a reduced SG of the overall models, when submodel-induced symmetries are present, will also be discussed. However, details about the *KO* driven composition scheme are omitted here, they can be found in [Sie95]. This section will be concluded by pointing out the limitations of Kronecker operator based composition schemes.

2.5.1 Fundamentals

In the following we briefly introduce some concepts as far as it is important for an understanding of the following paragraphs, a deeper discussion will follow in chapter 4.

It is assumed that each submodel S_i consists of a finite ordered set of discrete state variables (SVs) $\mathfrak{s}_j^i \in \mathfrak{S}^i$, and a finite set of activities ($\mathcal{A}ct_i$). By executing activities, one at a time, the model evolves from one state to another, where each SV $\mathfrak{s}_j^i$ can take an arbitrary value from $\mathbb{N}$. Since the set of SVs is finite and ordered, each state of submodel S_i can be written as a vector $\vec{s}_i$. The finite set of all reachable (submodel-local) states is denoted as $\mathbb{S}_i \subset \mathbb{N}^{|\mathfrak{S}_i|}$, An element of $\mathbb{S}_i$ is a mapping:

$$\mathcal{M}_i : (\mathfrak{s}_1^i, \ldots, \mathfrak{s}_{|\mathfrak{S}^i|}^i) \rightarrow \mathbb{N}^{|\mathfrak{S}^i|}, \tag{2.18}$$

Thus a state of the overall model can be described as a vector over all submodel-local vectors $\vec{s}_i$, so that the set of all SVs is given as union of the submodel local ones ($\mathfrak{S} := \bigcup_{i:=1}^{n} \mathfrak{S}_i$) and a state of the overall model as vector $(s_1^1, \ldots, s_{n_1}^1, \ldots, s_i^j, \ldots, s_1^n, \ldots, s_{n_i}^n)$. Analogously the set of all activities is given by $\mathcal{A}ct := \bigcup_{i:=1}^{n} \mathcal{A}ct_i$. For simplicity it is now assumed that the local SG of submodel S_i is denoted T_i and that it is generated on beforehand.

2.5.2 SG composition for pure interleaving

If no interaction among the submodels takes place one speaks of pure interleaving, i.e. the submodels, which are in fact (disjoint) partitions of the overall model are executed in parallel. On the level of the SG of the overall models this means:

$$\frac{\vec{s}_i \xrightarrow{a,\lambda} \vec{t}_i}{(\vec{s}_1,\ldots,\vec{s}_i,\ldots,\vec{s}_n) \xrightarrow{a,\lambda} (\vec{s}_1,\ldots,\vec{t}_i,\ldots,\vec{s}_n)}$$

For the meaning of the above notation one may refer to Appendix A.4 (p. 163). An example of pure interleaving is depicted in Fig. 2.2: SG T_4 is obtained by pure interleaving of the submodels. I.e. the non-interactive composition of T_1 and T_2 results in a pure interleaving of the a transitions as induced by the high-level model activities of the respective submodel S_1 and S_2.

As a result the SG T of the composed overall model (S) is the Cartesian product of the local SGs (T_i) of the submodels (S_i), commonly denoted as product SG (PSG).[3] I.e. in case of n submodels one has:

$$S := S_1 \|_\emptyset S_2 \|_\emptyset \cdots \|_\emptyset S_n \Rightarrow T := T_1 \times T_2 \times \cdots \times T_n.$$

Each of the local SGs induces a transition rate matrix R_i of a (submodel-local) CTMC C_i (cf. Eq. 2.16). These matrices can be combined via a Kronecker sum, yielding the transition rate matrix R of the CTMC underlying the overall model [Pla85]:

$$S := S_1 \| S_2 \| \cdots \| S_n \Rightarrow R := \bigoplus_{i:=1}^{n} R_i$$

2.5.3 SG composition for activity synchronization

For simplicity we will now ignore priorities and execution rates of activities to be synchronized. This is justified, since in the context of this thesis, we will develop a SG exploration scheme, where constructivity as known from SPAs is irrelevant. Consequently the preservation of properties of the submodel-local SGs is here not important and thus the handling of priorities and execution rates in a compositional style not necessary.

Composition scheme

When executing two submodels (S_1 and S_2) in parallel, a subset of the activities ($\mathcal{S}$) may have to be executed jointly, whereas other activities are executed in a pure-interleaved fashion. As a result, the SG T of the composed SG is a subset of the Cartesian product of the local SGs T_i of the submodels S_i:

$$S = S_1 \|_\mathcal{S} S_2 \|_\mathcal{S} \cdots \|_\mathcal{S} S_n \quad \Rightarrow \quad T \subseteq T_1 \times T_2 \times \cdots \times T_n.$$

As an example consider once again Fig. 2.2. The rate of transition a in SG T_3, which results from a synchronization of T_1 and T_2, is given by $\varphi(\lambda, \mu)$. In case of the SPA Tipp [HHK+98], the rate of a synchronized Markovian transition is given by $\varphi(\lambda, \mu) = \lambda \cdot \mu$. However, other policies for computing the resulting transition rate are also possible [Hil94b].

In the following, we describe how a *synchronization* of timed activities has to be handled on the level of the SGs: Let $\mathcal{S} \subseteq \mathcal{A}ct$ be the set of activities to be jointly executed, where the activities to be synchronized in the various submodels are assumed to carry the same label. The following cases needs to be handled:

(1) **Pure interleaved** execution ($a \notin \mathcal{S}$): This case was already discussed above.

(2) **Synchronized** execution ($a \in \mathcal{S}$):

$$\frac{\vec{s}_i \xrightarrow{a,\lambda_a} \vec{t}_i \wedge \cdots \wedge \vec{s}_j \xrightarrow{a,\lambda_a} \vec{t}_j}{(\vec{s}_1,\ldots,\vec{s}_i,\ldots,\vec{s}_j,\ldots,\vec{s}_n) \xrightarrow{a,\varphi} (\vec{s}_1,\ldots,\vec{t}_i,\ldots,\vec{t}_j,\ldots,\ldots \vec{s}_n)}$$

where φ is a function over the set of rates of the activities participating in the synchronization.

[3] In case of SGs the Cartesian product is obtained by simply building the cross-product of the sets of the (submodel-local) states and still maintaining the transitions between the combined state descriptors.

Kronecker operator driven composition of SGs
In case the submodels to be synchronized expose all the relevant information to the environment, it is possible to construct the transition rate matrix from the submodel-local transition rate matrices by applying a Kronecker operator based scheme. To do this, it is required to split each submodel-local transition rate matrix into a transition rate matrix referring to the transitions as induced by non-synchronizing activities (R_i) and into a set of transition rate matrices, where each refers to the transition induced by a specific synchronizing activity a ($R_{i,a}$). The (potential) transition rate matrix of the overall model can then be constructed as follows [Pla85]:

$$S := S_1 \|_{\mathcal{S}} S_2 \cdots \|_{\mathcal{S}} S_n \Rightarrow \tilde{R} := \bigoplus_{i:=1}^{n} R_i + \sum_{a \in \mathcal{S}} \omega_a \bigotimes_{i:=1}^{n} R_{i,a} \tag{2.19}$$

The parameter ω_a is a predefined weight, so that the rates of transitions referring to synchronized activities are weighted with ω_a. But one may also apply other strategies, the only thing from concern is that the resulting rate can either be somehow computed from the submodel-local rates or is a predefined constant. In this latter case ω_a gives this constant value, whereas the non-zero entries of the submodel-local transition rate matrices $R_{i,a}$ are replaced by ones. However, one may note that the transition rate matrix $\tilde{R}$ may contain entries for non-reachable states, since the cross-product building of the above scheme does not take the initial state of the overall model into account. However, if one knows the set of reachable states, such positions can be identified easily.

Synchronization or Kronecker operator driven decomposition of models
Due to the advantages of compositional SG construction as illustrated at the beginning of this section, it is not surprising that concepts known from compositional SG construction have found their way into the world of monolithic high-level model descriptions, such as GSPNs, etc.. Here a monolithic model is decomposed into partitions, where the activities manipulating SVs of more than one partition are split accordingly. For obtaining the model's overall SG from the submodel's local SGs, activity synchronization over the previously split activities takes place [CT96, HHMR97], so that a representation of the model's overall SG is obtained by applying Eq. 2.19. In the following such a strategy will be denoted as *Sync* driven decomposition. However it is important to note that the finding of a good partitioning is still an open question.

2.5.4 SG composition for sharing of SVs

In case of a compositional SG construction the sharing of SVs is more complex, i.e. for composing the SGs of the submodels, one must guarantee that one only combine submodel-local states having the same values in the position of the shared SVs. For doing this one may proceed as follows: Let the set of shared SVs be denoted $\mathcal{J}$, where $\mathcal{J} \subseteq \mathfrak{S}$ and let for simplicity the SVs to be shared among the different submodels have the same identifier, e.g. $\mathfrak{s}_j$. Analogously to the synchronization one may implement the *Join*-composition operator on the level of submodel SGs then as follows:

$$S := S_1 < \mathcal{J} > S_2 \cdots < \mathcal{J} > S_n \Rightarrow \tilde{T} := \sum_{i:=1}^{n} T_i \times \mathbb{1}_{\mathfrak{S} \setminus \mathfrak{S}_i}, \tag{2.20}$$

where $\mathbb{1}_{\mathfrak{S} \setminus \mathfrak{S}_i}$ is the identity function on the SVs of set $\mathfrak{S} \setminus \mathfrak{S}_i$. Similar to the composition of submodels via joint execution of activities, the above scheme gives one also the potential transition rate matrix ($\tilde{T}$). But knowing the set of reachable states, allows one to identify positions referring to un-reachable states and replacing all respective matrix entries with zeros, yielding the proper transition rate matrix T. However, one may already note, that the above scheme is in general not to be realized by the means of a Kronecker sum (see discussion below), so that the explicit storage of T may not be avoidable. But when employing symbolic storage techniques, the memory consumption of T is not an issue.

2.5.5 SG composition for replication of submodels

The *Rep*-operator can be combined with the sharing of SVs or an activity synchronization. In both cases it is possible to explore the submodel-local SGs and to apply one of the above composition schemes for obtaining an un-reduced transition rate matrix of the overall model and subsequently executes a reduction (a-posteriori reduction). However, when employing non-symbolic storage techniques, such a naive approach may be hampered by the size of the un-reduced transition rate matrix. Thus the advantage of an compositional SG reduction technique is at hand, it simply avoids the construction of the un-reduced transition rate matrix. But due to the nature of the *Join* and *Sync*-operator, *Rep**Join* and *Rep**Sync* have to be discussed separately.
Before we proceed, one may note that we only will roughly investigate compositional SG reduction, since in the context of this thesis an a-posteriori techniques will be introduced. This is justified, since due to our symbolic technique the generation and storage of un-reduced SGs is not an issue.

Compositional SG reduction when the Rep\Sync-*operator is employed*

The *Rep*-operator in combination with the synchronization of activities can be employed for constructing a reduced SG in a compositional style. How to do this is described in [Sie95]. The main idea is as follows: At first one construct a reduced SG for the model consisting of the replicated submodels only. Secondly one applies a Kronecker operator driven composition scheme for composing the reduced SGs of the replicated submodels with the submodel-local SG of the non-symmetric units, yielding a representation of a reduced transition rate matrix of the overall model.

Compositional SG reduction when the Rep\Join-*operator is employed*

Contrary to activity-synchronization the *Join*-method does not allow to apply the concept of *constructivity* in a straight forward manner. The notion of equivalence must be modified in such a way, that two submodels are equivalent not only if they have the same timed behavior, but also expose the same set of values taken by the SVs to be shared, to the environment. Also the abstraction of internal behavior may only be restricted to activities not affecting SVs, which are employed for sharing. If these aspects are obeyed, it seems to be straight forward to reduce the SG of the replicated submodels and subsequently apply the *Join*-composition scheme as introduced above. However, to the best of our knowledge such a procedure is not yet described in the literature.

2.5.6 Limitation of Kronecker operator driven composition schemes

There are two significant drawbacks attached to schemes employing compositional SG construction and representation. (a) The schemes either require the generation of the SGs of the submodels in isolation and/ or (b) they require the high-level model to be of a certain structure. This makes the employment, especially of the decomposition techniques in practice often problematic, if not impossible.

For coping with the first problem, recently developed approaches, including our own activity/reward-local scheme, explore the submodels in a submodel-interdependent fashion. As consequence, upper bounds of SVs need not to be known in advance, nor does a submodel-local exploration in isolation have to be possible.

In contrast to the first (more general) problem, the second problem is not only related to a compositional SG construction procedure, but to the employed composition scheme (cf. Eq. 2.19 and 2.20). As it will be shown latter for applying a *KO* driven composition scheme it is required that the SVs of the submodels appear in a non-interrupted sequence. If such an order exists, the respective model is said to have a KO compliant structure (cf. Sec. 3.5.2, p. 62ff)
If one takes now into consideration that each submodel must contain at least all those SVs

which are relevant for the execution function of the submodel's activities, it is evident that there are high-level models which can not be transformed into their low-level representation by following a *Sync* driven decomposition strategy, since a *KO* compliant partitioning of them does simply not exists. In such cases a *KO* based composition scheme and all approaches making use of it are not applicable. In the context of compositionally constructed high-level models, this is most likely to be the case for *Join*-composed high-level models. However, for compositional constructed high-level models, where submodels are composed via *Sync*, this is not an issue, since in such setting a *KO* compliant structure of the overall high-level model is always present, due to the fact that the submodels do not share any of their SVs.

3

Zero-suppressed Multi-terminal BDDs: Concepts, Algorithms and Applications

As major contribution this chapter presents zero-suppressed Multi-terminal Decision Diagrams (ZDDs) and the concept of partially-shared ZDDs (*p*ZDDs). This new type of Decision Diagram will be shown to give canonical representations for pseudo-boolean functions. Based on Bryant's BDD-algorithms, algorithms for efficiently manipulating *p*ZDDs will be developed. A survey of ZDD based set and matrix representation and manipulation will round this chapter, where concepts known from Algebraic Decision Diagrams (ADDs) for solving sets of linear and differential equations are extended to the case of ZDDs.

3.1 Organization of the chapter

In order to simplify the definitions of the different types of Binary Decision Diagrams (BDDs), this chapter obeys a hierarchy of inheritance of basic concepts among the various types (cf. Fig. 3.15, p. 70). Since the different data structures become more complex as more different concepts are combined, this chapter starts in Sec. 3.2 with the ordinary and most simple form, the Binary Decision Trees (BDT). After the reader is familiar with the basic concepts, the next thing to follow is the introduction of reduction rules. Therefore Sec. 3.2 subsequently introduces the basic reduction rules for BDTs, yielding *isomorphism-free* and *shared* BDDs, and their extended version of *don't-care-free* and *zero-suppressed* BDDs. The discussion on the basic BDD-types is rounded by Sec. 3.2.3 introducing their multi-terminal extensions. This yields the well known type of (*don't-care-free*) Multi-terminal BDDs, which are also commonly denoted as Algebraic Decision Diagrams (ADDs) and *zero-suppressed* Multi-terminal BDDs (ZDDs), where to the best of our knowledge this latter type has not been described in the literature yet.

If employing the zero-suppressing reduction rule, the graph of a DD alone does not define the represented function, the set of boolean variables the DD is defined over also must be known. Within a shared BDD environment it is required to handle ZDDs being defined over different sets of boolean variables. Standard algorithms as known from literature fail to do so. For solving this problem, this work develops the concept of *partially shared* ZDDs (*p*ZDDs), which is described in Sec. 3.3. Based on this concept, Sec. 3.4 introduces algorithms for efficient manipulation of *p*ZDDs.

Now the discussion is ready for tackling the main issue of symbolic data types in the context of this work: Symbolic representation of sets, transition relations and matrices. Therefore Sec. 3.5 discusses how *z*-BDDs over n boolean variables and their multi-terminal extensions can be employed for encoding sets of states, - transition relations and real-valued matrices. Since the solvers for computing state probabilities considered in this work follow the hybrid solution approach, "pure"-symbolic matrix-matrix - and matrix-vector operations are irrelevant, except the cross-product building of matrices. Due to its relevance for SG composition (cf. Sec. 2.5, p. 25ff), this operation in case of ZDD based matrices and its difference to the *Kronecker product* of matrices is discussed in great detail. What follows next is a discussion on how to extend ZDDs for efficiently solving very large sets of linear and differential equations, where we follow the hybrid approach as suggested by [Par02] for ADDs. The last paragraph of Sec. 3.5 is then devoted to briefly discussing the latest storage techniques going beyond symbolic matrix representations.

The chapter is concluded by Sec. 3.6, which gives an overview over related work and indicates our own contributions.

3.2 Binary Decision Diagrams and extensions

3.2.1 Binary Decision Diagrams (BDDs)

Binary Decision Trees (BDTs)
An ordered Binary Decision Tree (BDT) is a bi-partite graph, where the variables labeling the non-terminal nodes obey a fixed total ordering. BDTs are defined as follows:

Definition 3.1: Binary Decision Tree

An ordered Binary Decision Tree (BDT) $\mathsf{B} < \mathcal{V}, \pi >$ is a rooted binary tree $\mathsf{B} < \mathcal{V}, \pi >:= \{\mathcal{K}, \texttt{var}, \texttt{then}, \texttt{else}\}$:

(1) $\mathcal{V}$ is a finite and non-empty set of boolean variables with the fixed ordering relation $\pi \subseteq \mathcal{V} \times \mathcal{V}$ defined one.

(2) $\mathcal{K} = \mathcal{K}_T \cup \mathcal{K}_{NT}$ is a finite non-empty set of nodes, consisting of the set of terminal nodes $\mathcal{K}_T$ and non-terminal nodes $\mathcal{K}_{NT}$, with $\mathcal{K}_T \cap \mathcal{K}_{NT} = \emptyset$.

(3) The following functions are defined:
  (3.a) the value-returning function $\texttt{value} : \mathcal{K}_T \mapsto \mathbb{B}$ for each terminal node,
  (3.b) the variable-returning function $\texttt{var} : \mathcal{K}_{NT} \mapsto \mathcal{V}$ for each non-terminal node,
  (3.c) the child node-returning functions $\texttt{else}, \texttt{then} : \mathcal{K}_{NT} \mapsto \mathcal{K}$ for each non-terminal node, and
  (3.d) the root node-returning function $\texttt{getRoot} : \mathsf{B} \mapsto \mathcal{K}$.

(4) For the BDT to be ordered the following constraint must hold:

$$\begin{aligned} &\forall u \in \mathcal{K}_{NT} : \\ &\qquad \texttt{then}(u) \in \mathcal{K}_{NT} : \texttt{var}(\texttt{then}(u)) \ {}_{\pi}{>} \ \texttt{var}(u) \\ &\qquad \texttt{else}(u) \in \mathcal{K}_{NT} : \texttt{var}(\texttt{else}(u)) \ {}_{\pi}{>} \ \texttt{var}(u). \end{aligned}$$

The relation π allows one to write the elements of $\mathcal{V}$ as a vector $\vec{\mathsf{v}} := (\mathsf{v}_1, \ldots \mathsf{v}_{n_\mathcal{V}})$. It is defined that v_1 is the boolean variable with the lowest and $\mathsf{v}_{n_\mathcal{V}}$ the one with the highest order with respect to the elements of $\mathcal{V}$. We define that $\vec{\mathsf{v}}[i]$ returns the variable at position i, where we will also write v_i instead. Since π is assumed to be fixed, we will also employ the notation $\mathsf{B} < \vec{\mathsf{v}} >$, in case $\vec{\mathsf{v}}$ is also known we simply write B.

While traversing a BDT from the root node to a terminal node, one visits a sequence of nodes. Such a sequence $p := (n_r, \ldots, n_q, t)$; with $n_r, n_q \in \mathcal{K}_{NT}$ and $t \in \mathcal{K}_T$ and is commonly denoted as path. In the following $\mathbb{P}_\mathsf{B}$ denotes the set of all paths of a BDT B. The nodes as contained in paths are ordered by the sequence of their appearance, starting at the root node of the BDT.

Summarizing the above discussion, the following conventions, concerning the graphical representation of BDTs and their derivatives, are applicable:

(1) The nodes of a BDT are organized level-wise, where all nodes labeled with the same variable $x \in \mathcal{V}$ appear at the same level. Sometimes we therefore speak of a level of a node, rather than of its variable.

(2) The order π is applied from top to bottom, so that the variable with the highest order appears at the bottom, and the variable with the lowest order at the top-level.

(3) The terminal nodes always appear at the bottom-level.

(4) Arrow-heads of the directed edges among the BDT nodes can be omitted, since the graph of a BDT is to be traversed always from the root to the terminal node, i.e. from top to bottom.

(5) `then`− and `else`−edges, also commonly denoted as 1- and 0-edge, are represented by a solid, dashed line respectively. In this respect one also speaks of a 0- or 1-child of a node, when referring to the nodes reached via the respective outgoing edge of a non-terminal node.

Fig. 3.1.i (p. 34) illustrates a complete BDT, as one can see the variable ordering is obeyed on all paths. The non-terminal node n_3 is the 1-child of root node n_1, where the 0-child of the latter is node n_2. One may note that in contrast to Fig. 3.1, the above definition do not require the BDTs to be complete. I.e. a path within a BDT may not necessarily contain all of its function variables.

Semantics of BDTs

The Shannon expansion develops a boolean function $f : \mathbb{B}^{|n_\mathcal{V}|} \rightarrow \mathbb{B}$ by replacing each of the $n_\mathcal{V}$ variables of $\mathcal{V}$ by the respective literal (cf. Def. A.1, p. 162). It is straight forward to organize the process of BDD construction for a boolean function by applying the Shannon-expansion recursively, starting with the allocation of nodes at the level of terminal nodes. Thus it follows immediately that a BDT must be the graph based representation of a boolean function. Consequently each BDT-node represents a function. The set of boolean assignments fulfilling a boolean function f^k represented by a BDT-node k can be defined as follows:

$$Sat_k := \{\vec{b} \,|\, \texttt{Satisfy}(\vec{b}, k, \vec{\mathsf{v}}) \neq 0\} \tag{3.1}$$

where `Satisfy` is specified as Algo. B.1 (p. 164). As input parameters it takes a bit string of length $n_\mathcal{V}$, the root node of the respective BDT and the vector of function variables the BDT is defined over. Each of the fulfilling assignments $\vec{b}$ defines a monomial as follows:

$$\mathcal{M}interm(\vec{\mathsf{v}}, \vec{b}\,) := \bigwedge_{j:=1}^{n_\mathcal{V}} x_j, \text{where } x_j := \begin{cases} \mathsf{v}_j & iff\, b_j = 1 \\ \neg\mathsf{v}_j\, iff\, b_j = 0 \end{cases} \tag{3.2}$$

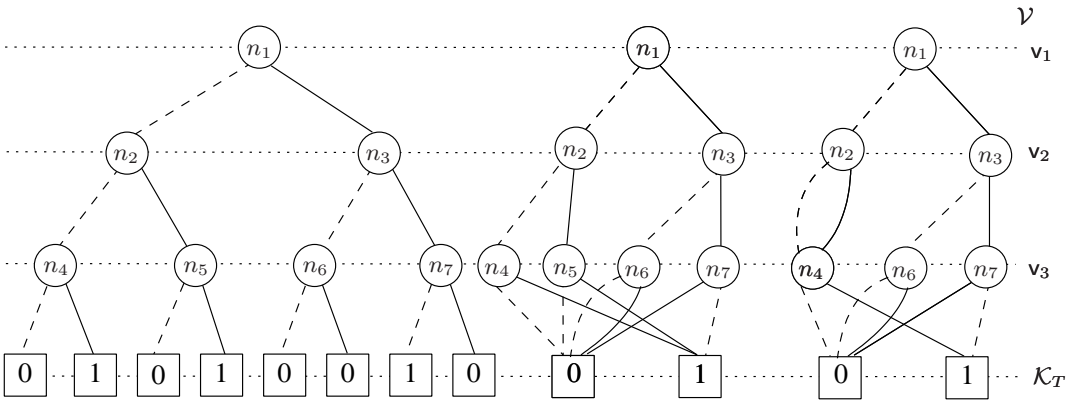
where the j'th variable is mapped to the j'th bit position. Since each induced monomial is complete with respect to the set of boolean variables the BDT rooted in n is defined over, one may also speak here of minterms as defined by Eq. 3.2. The disjunction over all the induced minterms gives one the canonical SOP of the represented boolean function as follows:

$$\bigvee_{Sat_n} \mathcal{M}interm(\vec{\mathsf{v}}, \vec{b}\,) \tag{3.3}$$

From the above discussion it arises that from the graph of a BDT directly a boolean function can be deduced. Furthermore each BDT node n already defines a boolean function $f^n(\vec{b}\,)$, where the mapping of bit position i to a variable $\mathsf{v}_j \in \mathcal{V}$ must be adjusted accordingly ($\mathsf{v}_j = \vec{\mathsf{v}}[i]$). Function f^k may also be denoted f^B if k is the root node of BDT B. The terminal *0-node* gives the constant 0-function and the terminal *1-node* the constant 1-function.

Isomorphism-free BDDs

If one turns back to the BDT of Fig. 3.1.i, it is easy to see that the terminal *1-node*s and *0-node*s occur multiple times. This redundancy can be eliminated by representing all these terminal nodes by a single *1-node* and a single *0-node*, where the incoming arcs of the nodes to be replaced are re-directed towards the remaining nodes (see Fig. 3.1 (i) and (ii)). As a result, one obtains a degenerated BDT, where the non-terminal nodes of a BDT with



(i) Complete BDT (ii) Partially reduced BDD (iii) *iso-free* BDD

Figure 3.1: BDTs and the merging of isomorphic structures

the highest order, which are labeled by the same variable, may possess the same 1- and 0-child (terminal) nodes. Such non-terminal nodes are isomorphic, and they represent the same boolean function, including variable labeling. Consequently such nodes can be merged, where the redundant one can be eliminated once again. I.e. in general within a BDD-environment, BDTs representing the same boolean function are merged. The merging of two BDT-nodes can hereby be recursively applied in a bottom-up manner, or it can be directly incorporated into the node allocating function. The isomorphism of nodes is defined as follows:

Definition 3.2: Isomorphism for BDT nodes

Two BDT-nodes $u, v \in \mathcal{K}$ of a BDT B are isomorphic if they represent the same boolean function, including the labeling of the employed variables. This yields the following recursive definition:

(1) Non-terminal case: $u, v \in \mathcal{K}_{NT}$

$$u \equiv v \Leftrightarrow \mathtt{var}(u) = \mathtt{var}(v) \ \wedge \ \mathtt{else}(u) = \mathtt{else}(v) \ \wedge \ \mathtt{then}(u) = \mathtt{then}(v)$$

(2) Terminal case: $u, v \in \mathcal{K}_T$: $u \equiv v \Leftrightarrow \mathtt{value}(u) = \mathtt{value}(v)$

The application of the above definition enforces a collapsing of isomorphic sub-structures, so that the result is a directed acyclic graph rather than a tree, which is why one speaks of Binary Decision *Diagrams*. An isomorphism-free Binary Decision Diagram (*iso-free* BDD) is defined as follows:

Definition 3.3: Isomorphism-free Binary Decision Diagram

An isomorphism-free BDD (*iso-free* BDD) $\mathsf{B} < \mathcal{V}, \pi >$ is a degenerated BDT $\mathsf{B} := \{\mathcal{K}, \mathtt{var}, \mathtt{then}, \mathtt{else}\}$, where in the sense of Def. 3.2 only unique nodes exist:

$$\nexists\, u, v \in \mathcal{K} : u \neq v \wedge \ u \equiv v.$$

In the following **all BDDs are considered to be isomorphism-free.**

Fig. 3.1 shows how the merging of isomorphic nodes can be applied in a bottom-up fashion. However, this is not necessary, one simply needs to encapsulate the above rule into the functions: $\mathtt{getTerminalNode}(b)$, where $b \in \{0, 1\}$ and $\mathtt{getUniqueNode}(\mathsf{v}, t, e)$, where $\mathsf{v} \in \mathcal{V}$

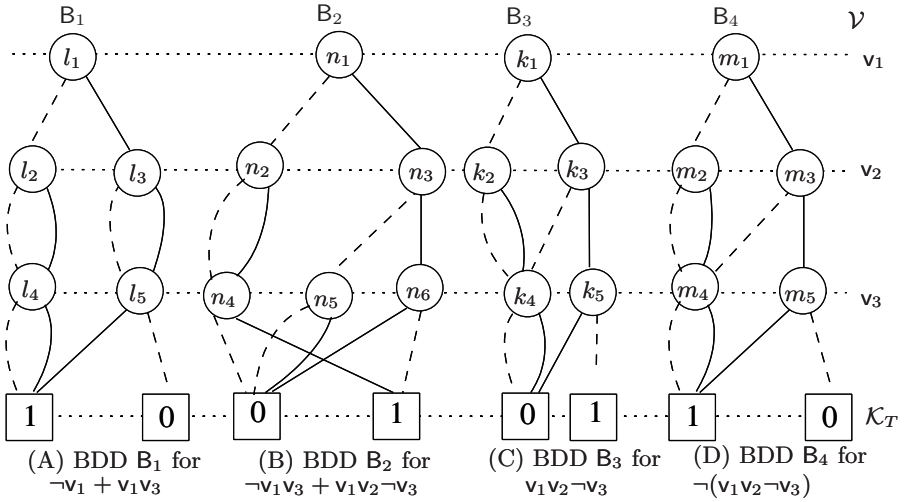


Figure 3.2: *iso-free* BDD based representations of boolean functions

and $t, e \in \mathcal{K}$ for allocating terminal and non-terminal nodes within a BDD-environment. These functions allocate a new node only if there does not yet exists an isomorphic one, otherwise they return the existing node. This strategy is very important, when handling different BDDs at the same time, since even among different BDDs it guarantees, that isomorphic nodes are merged, which yields shared or multi-rooted BDDs, as to be discussed next. Examples of *iso-free* BDDs are drawn Fig. 3.2.

Shared BDD-environment

The concept of sharing (sub-)graphs between different BDDs is straight forward and a well accepted technique to increase the efficiency of BDDs. Due to the strict application of the isomorphism rule of Def. 3.2 (p. 34) within a single BDD-environment, two isomorphic nodes never co-exist. Consequently within a shared BDD-environment there is in principle only one global set of nodes for each variable, one global set of terminal nodes and one global set of boolean variables ($\mathcal{V}$). However, the set of function variables of a function f^n, as represented by node n is a sub-set of $\mathcal{V}$ ($\mathcal{V}^n \subseteq \mathcal{V}$), requiring the adaptation of the mapping of bit position i to variable $\mathsf{v}_j \in \mathcal{V}^n$. However, concerning function `Satisfy` this is irrelevant, since its third input parameter is assumed to be the ordered sequence of the function variables of the represented function (cf. Algo B.1).

In the remainder of this section, all BDDs are assumed to be *shared*. But in terms of graphical representation the concept of non-shared BDDs is maintained, which improves the clearness of the drawn graphs as well as the clearness of the call-trees of the algorithms presented later. Fig. 3.3 shows the respective graph if the BDDs of Fig. 3.2 were allocated in a shared BDD-environment.

Don't-care-free BDDs

A *don't-care-free* BDD (dnc-*free* BDD) is a BDD, where don't-care nodes are eliminated, and the incoming edges are re-directed to the child n. Hereby a non-terminal node d is considered as *dnc*-node if its 1- and 0-edge point to the same node $n \in \mathcal{K}$. The Shannon expansion concerning a function f as represented by a BDD rooted in a *dnc*-node labeled with variable v_k is given as follows:

$$\begin{aligned} f &= \neg\mathsf{v}_k f_0 + \mathsf{v}_k f_1 \text{ where } f_0 = f_1 \text{ per } \textit{dnc}\text{-rule} \\ &= (\neg\mathsf{v}_k + \mathsf{v}_k) f' \text{ for } f' := f_0 = f_1 \\ &= f' \end{aligned} \tag{3.4}$$

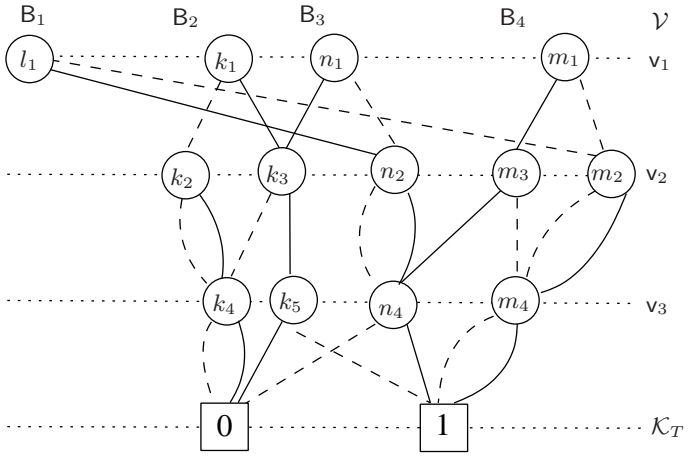


Figure 3.3: Shared or multi-rooted *iso-free* BDDs

Obviously variable v_k is a non-decisive variable for function f^d. The sub-functions f_1 and f_0, which are rooted in the `then`- and `else`-child n of don't-care node d are the 1- and 0 co-factor of function f^d, consequently we will also use the notation f_1^d and f_0^d. A dnc-*free* BDD can be formally defined now as follows:

Definition 3.4: Don't-care-free BDD

A *don't-care-free* BDD (dnc-*free* BDD) $\mathsf{B} < \mathcal{V}, \pi >$ is a *iso-free* BDD $\mathsf{B} := \{\mathcal{K}, \texttt{var}, \texttt{then}, \texttt{else}\}$ where *don't care nodes* are eliminated:

$$\nexists u \in \mathcal{K}_{NT} : \texttt{else}(u) = \texttt{then}(u).$$

For exemplification one may turn to Fig. 3.4, which shows diverse examples of *dnc-free* BDDs as obtained from the BDDs of Fig. 3.2 (p. 35) by eliminating the *dnc*-nodes bottom-up and merging the newly created isomorphic sub-structures. However, it should be clear that the *dnc*-reduction rule can easily be encapsulated into a node allocating function. Thus it is not necessary to define a respective reduction algorithm for obtaining dnc-*free* BDDs. This node-allocating algorithm (`getUniqueNode`) returns a node labeled with variable v_i if $e \neq t$. In case $e = t$ `getUniqueNode` returns node e, since under the *dnc*-reduction rule incoming arcs of a *dnc*-node are re-directed to its 0 and 1-child.

In cases of paths where levels are skipped, more than one assignment is deducible. This feature arises from the fact that for each skipped-variable v_j, which we will also denote as *dnc*-variable, the respective bit at position i can take the values 0 *or* 1. As a consequence each path, containing the terminal *1-node* induces ultimately two minterms for each *dnc*-variable v_j, namely a minterm where the variable x is set to v_j and a minterm where x is set to $\neg\mathsf{v}_j$. But the existence of *dnc*-variables does not require a re-definition of algorithm `Satisfy`, namely the version introduced for BDTs handles paths where variables are skipped (cf. line 3 of Algo. B.1, p. 164). According to Eq. 3.3 (p. 33) one may build the disjunction over all minterms as induced by the set of fulfilling assignments, in order to obtain the canonical SOP of the represented function. It is straight-forward that *dnc*-variables can be eliminated then from each minterm, so that one yields the minimal DSOP of the represented function. Thus it should be clear that if $\mathcal{V}$ and π is fixed, a dnc-*free* BDD is a (strong) canonical representation of a boolean function, where a graph alone defines the represented function (cf. [Sas96, MT98, Bai05]).

In a shared BDD-environment a *dnc*-reduction rule also implies that for a BDD B it does not matter, whether it is defined on the whole set of boolean variables or not. I.e.

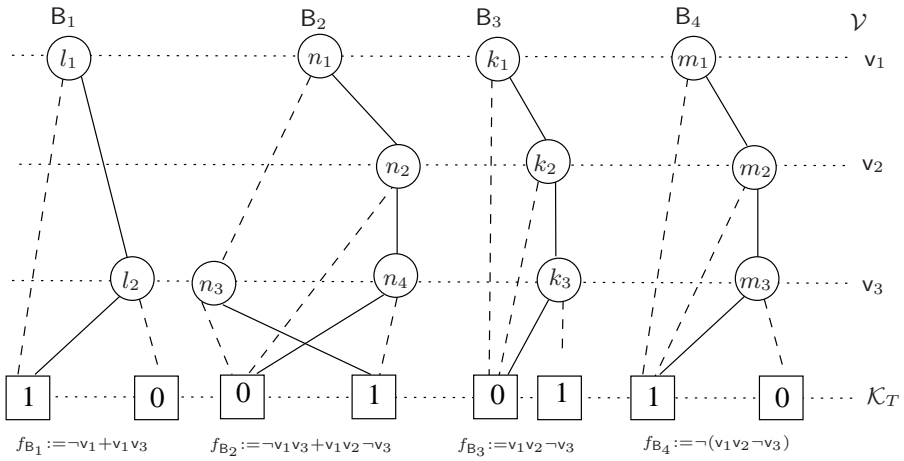


Figure 3.4: *dnc*-free BDD based representations of boolean functions

all variables $\notin \mathcal{V}^{\mathsf{B}}$ are skipped within all paths as contained on $\mathbb{P}_{\mathsf{B}}$. Due to their *dnc*-semantics **on all paths**(!), these variables are not-decisive for the represented function. For exemplification one may refer to B_1 as depicted in Fig. 3.4. Since the level of variable v_2 is skipped on all paths leading to the terminal *1-node* $(l_1, 1), (l_1, l_2, 1)$ one may remove v_2 from the set of variables of B_1. I.e. according to Eq. 3.3 (p. 33) one obtains $f(\mathsf{v}_1, \mathsf{v}_2, \mathsf{v}_3) := \neg\mathsf{v}_1\mathsf{v}_2 + \neg\mathsf{v}_1\neg\mathsf{v}_2 + \mathsf{v}_1\mathsf{v}_2\mathsf{v}_3 + \mathsf{v}_1\neg\mathsf{v}_2\mathsf{v}_3$ which can be re-formulated to $f(\mathsf{v}_1, \mathsf{v}_3) := \neg\mathsf{v}_1 + \mathsf{v}_1\mathsf{v}_3$, which truly does not depend on the variable v_2.

In the literature *dnc-free* BDDs are often denoted as RO BDDs or simply as BDDs. In order to simplify the definition of zero-suppressed BDDs, a more restrictive notation is emphasized. I.e. in the following a **BDD addresses the isomorphism-free version of an ordered BDD**, where in case of a **dnc-*free* BDD the *dnc*-reduction rule is additionally obeyed**.

3.2.2 Zero-suppressed BDDs (*z*-BDDs)

Zero-suppressed BDDs [Min93] are derivatives of BDDs, however all non-terminal nodes, where the 1-child is the terminal 0-node (*0*-suppressed node), are eliminated. Consequently the expansion rule concerning a function f as represented by a BDD rooted in a *0*-suppressed-node labeled with variable v_k is given by:

$$\begin{aligned} f &= \neg\mathsf{v}_k f_0 + \mathsf{v}_k f_1 \text{ where } f_1 = 0, \text{ per } \textit{0-sup}\text{-rule} \\ &= \neg\mathsf{v}_k f_0 \end{aligned} \tag{3.5}$$

According to the Shannon expansion f is obviously not independent of v_k, thus v_k is one of its decisive variables! In order to eliminate *0-sup.*-nodes, their incoming edges are re-directed towards the 0-child. The *0-sup.* reduction applied in a recursive manner yields:

Definition 3.5: Zero-suppressed BDD

A zero-suppressed BDD (*z*-BDD) $\mathsf{Z} < \mathcal{V}, \pi >$ is a *iso-free* BDD $\mathsf{B} := \{\mathcal{K}, \texttt{var}, \texttt{then}, \texttt{else}\}$, where

$$\nexists\, u \in \mathcal{K}_{NT} : \texttt{then}(u) = \textit{0-node}.$$

Fig. 3.5 shows some examples for *z*-BDDs, where the drawn graphs are obtained by applying the 0-suppressing (*0-sup.*) reduction rule to the BDDs of Fig. 3.2 (p. 38). Analogously to the *dnc*-, the *0-sup.*-reduction rule can be easily encapsulated in the respective node allocating function (`getUniqueZDDNode`). This function returns a node labeled with variable v_i if $t \neq$ *0-node*, otherwise it returns node e, since under the *0-sup.*-reduction rule incoming

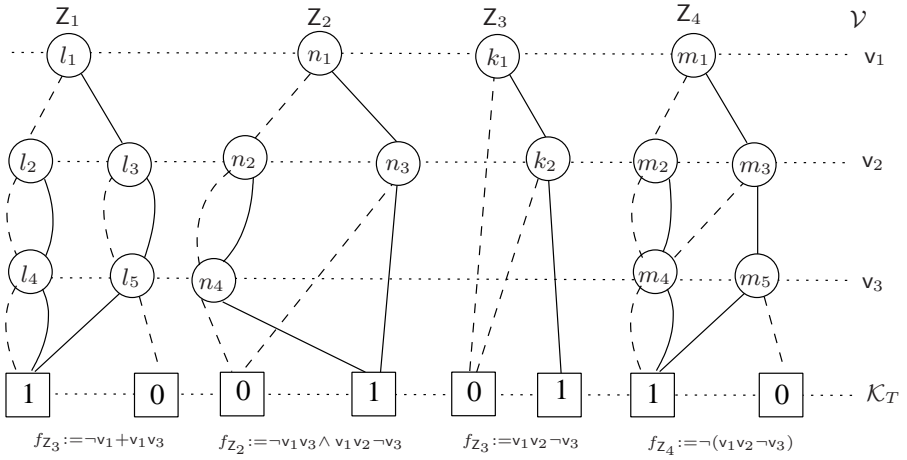


Figure 3.5: *z*-BDD based representations of boolean functions

arcs of a *0-sup.* node are re-directed to its 0-child.

As a consequence of the *0-sup.* reduction rule (cf. Eq. 3.5, p. 37) the graph of a *z*-BDD alone does not define the represented boolean function. One also needs to know the set of variables of the *z*-BDD, in order to deduce the encoded boolean function correctly. For exemplification one may refer once again to the *z*-BDD Z_3 of Fig. 3.5. Here variable v_3 is skipped on all paths within Z_3. Under a dnc-*free* BDD this does not matter, since v_3 is non-decisive, i.e. it is interpreted as being either 1- or 0-assigned, so that the value of the bit at position 3 does not change the function value. Thus the graph interpreted as a dnc-*free* BDD defines the boolean function $f := \mathsf{v}_1\mathsf{v}_2\mathsf{v}_3 + \mathsf{v}_1\mathsf{v}_2\neg\mathsf{v}_3 = \mathsf{v}_1\mathsf{v}_2$. If one interprets the graph as a *z*-BDD, the situation for paths ending up in the terminal *1-node* differs. If one assumes that $\mathcal{V}_{\mathsf{Z}_3} := \{\mathsf{v}_1, \mathsf{v}_2\}$ the same situation as in case of a *dnc*-free BDD is achieved, namely the value of v_3 does not change the outcome of function $f_{\mathsf{Z}_3}(\mathsf{v}_1, \mathsf{v}_2)$. However, since $\mathcal{V}_{\mathsf{Z}_3} := \{\mathsf{v}_1, \mathsf{v}_2, \mathsf{v}_3\}$, only the assignment $\vec{b} := 110$ yields a function value of 1, where all other assignments can be evaluated to the function value 0. According to Eq. 3.3 (p. 33) this yields the function $f_{\mathsf{Z}_3}(\mathsf{v}_1, \mathsf{v}_2, \mathsf{v}_3) := \mathsf{v}_1\mathsf{v}_2\neg\mathsf{v}_3$ instead of $\mathsf{v}_1\mathsf{v}_2$. As one can see, in contrast to BDDs and *dnc-free* BDDs, for a *z*-BDDs graph the represented function may change as soon as the set of variables changes. E.g. within a shared BDD-environment the terminal *1-node* in isolation must be interpreted as the function $\bigwedge_{i:=1}^{n_\mathcal{V}} \neg\mathsf{v}_i$, rather than the constant 1-function as in case of BDDs and dnc-*free* BDDs. In contrast, the constant 0-function is still represented by the terminal *0-node*, no matter which BDD type is employed. So in case of *z*-BDDs, a shared BDD environment means not only the sharing of the same sub-graphs, it means also a sharing of all variables the *z*-BDDs are defined over. If $\mathcal{V}$ and π are fixed, a *z*-BDD $\mathsf{Z} < \mathcal{V}, \pi >$ is a (weak) canonical representation of a boolean function [Min96, Bai05] and its graph rooted in node n gives the canonical SOP of this function f_n by applying Eq. 3.3 in combination with the appropriate version of function `Satisfy`. The *0-sup.*-version of the latter is presented as Algo. B.2 (p. 164). **For the time being it is therefore now assumed, that the *z*-BDDs are all defined on the same set of variables $\mathcal{V}$.**

3.2.3 Multi-terminal BDDs (ADDs)

Don't-care free Multi-terminal Binary Decision Diagrams, which are commonly known as Multi-terminal Binary Decision Diagrams (MTBDDs) or Algebraic Decision Diagrams (ADDs), are an extension of *dnc-free* BDDs [ADD97]. The original concept was introduced in 1993 by two different groups of authors [CFM+93, CMF+93, BFG+93]. Extending a dnc-*free* BDD to the case of an ADD is straight forward. One simply extends the co-domain of function `value` (cf. Def. 3.1, p. 32) to a finite set $\mathbb{D}$ from an arbitrary universe, e.g. $\mathbb{R}$. Consequently an ADD M is the graph based representation of a pseudo-boolean function f_M: $\mathbb{B}^n \mapsto \mathbb{D}$, and we formally define it as follows:

(i) Function f_{B}

$\mathsf{v}_3\mathsf{v}_4\mathsf{v}_5\mathsf{v}_6$	$f_B(\vec{b})$
$b_1b_2b_3b_4$	
1 1 1 0	δ
1 0 0 1	
0 1 0 1	γ
0 1 0 0	
0 0 1 0	μ

(ii) Function f_{A}

$\mathsf{v}_1\mathsf{v}_2$	$f_A(\vec{b})$
b_1b_2	
1 1	1
0 0	1

(iii) ADDs representing function f_{A} and f_{B}

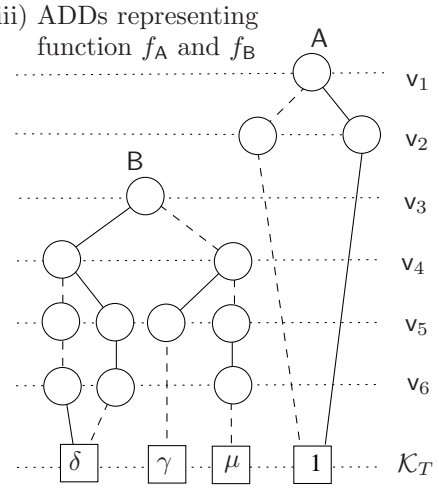


Figure 3.6: ADD based representations of pseudo-boolean functions

Definition 3.6: Algebraic Decision Diagram

An Algebraic Decision Diagram (ADD) $\mathsf{M} < \mathcal{V}, \pi, \mathbb{D} >$ is a dnc-*free* BDD $\mathsf{B} := \{\mathcal{K}, \texttt{var}, \texttt{then}, \texttt{else}\}$, where

(1) a finite non-empty set $\mathbb{D}$,

(2) and a function $\texttt{value} : \mathcal{K}_T \mapsto \mathbb{D}$ is defined.

The functionality of allocating terminal nodes holding also values other than 0 and 1, can be easily encapsulate into function `getTerminalNode`. Since the set of variables $\mathcal{V}$ and its total order π are assumed to be fixed and since the set of terminal values is often not from concern, we will also employ the notation M.
Examples for ADD based representations of pseudo boolean functions are given in Fig. 3.6, where the ADD A is the special case of a dnc-*free* BDD, also commonly denoted as 0-1 ADD. For clarity the labels of the nodes were omitted, as well as the edges leading to the terminal *0-node*. These "graphical simplifications" will be maintained throughout the rest of this work, unless required for exemplification purpose. In order to reduce the function tables as presented in Fig. 3.6.i and ii, the non-decisive variables of the represented functions are omitted, e.g. $\mathsf{v}_1, \mathsf{v}_2 \notin \mathcal{V}^{\mathsf{B}}$. Furthermore, table i and ii show also the mapping of each boolean variable v_j to its bit positions b_i within the assignment fulfilling the function f_{A}, f_{B} respectively.

3.2.4 Zero-suppressed Multi-terminal BDDs (ZDDs)

Like *dnc-free* BDDs were extended to ADDs, one may also extend *z*-BDDs to the multi-terminal case, yielding:

Definition 3.7: Multi-terminal zero-suppressed BDD

A multi-terminal zero-suppressed BDD (ZDD) $\mathsf{Z} < \mathcal{V}, \pi, \mathbb{D} >$ is a *z*-BDD $\mathsf{Z} := \{\mathcal{K}, \texttt{var}, \texttt{then}, \texttt{else}\}$, where

(1) a finite non-empty set $\mathbb{D}$,

(2) and a function $\texttt{value} : \mathcal{K}_T \mapsto \mathbb{D}$ is defined.

(i) Function f_{B}

v_1	v_2	v_3	v_4	v_5	v_6	$f_B(\vec{b})$
b_1	b_2	b_3	b_4	b_5	b_6	
dnc	*dnc*	1	0	0	1	δ
dnc	*dnc*	1	1	1	0	
dnc	*dnc*	0	1	0	1	γ
dnc	*dnc*	0	1	0	0	
dnc	*dnc*	0	0	1	0	μ

(ii) Function f_{A}

v_1	v_2	v_3	v_4	v_5	v_6	$f_A(\vec{b})$
b_1	b_2	b_3	b_4	b_5	b_6	
1	1	*dnc*	*dnc*	*dnc*	*dnc*	1
0	0	*dnc*	*dnc*	*dnc*	*dnc*	1

(iii) Shared ZDDs for f_{A} and f_{B}

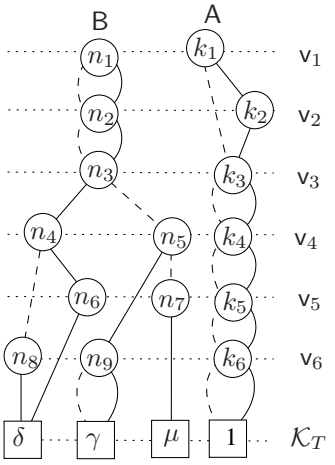


Figure 3.7: ZDD based representations of pseudo-boolean functions

Examples for ZDD based representations of pseudo-boolean functions are given in Fig. 3.7. The illustrated ZDDs represent herby the same pseudo-boolean functions as already illustrated in Fig. 3.6. However, as one can see, various *dnc* nodes are required to be inserted, n_1, n_2, n_9 and $k_3 - k_6$. Since some of them appear on all paths, the associated boolean variables are non-decisive variables concerning the represented function (cf. Eq. 3.4, p. 35). In order to reduce the size of the function tables as shown in Fig. 3.7.i and 3.7.ii, bit positions referring to non-decisive variables are marked accordingly, so that their explicit 0- and 1-assignments could be omitted. Furthermore table i and ii show also the mapping of each boolean variable v_j to its bit positions b_i within the assignment fulfilling the function f_{A}, f_{B} respectively. It is clear that one may wish to remove non-decisive variables and their *dnc*-nodes, not only from the function table, but also from a ZDD's graph, e.g. $\mathsf{v}_1, \mathsf{v}_2$ and $\mathsf{v}_3 \notin \mathcal{V}^{\mathsf{B}}$ and thus from the set of boolean variables it is defined over. However, until now a (fully) shared BDD-environment is assumed, and thus *dnc* nodes at the respective levels need to be allocated, since otherwise one would interpret the associated bit position –referring to skipped variables– as 0-assigned!

3.3 Partially shared ZDDs (*p*ZDDs)

Until now, it was assumed that all BDDs were allocated in a shared BDD-environment, i.e. *all* decision diagrams (DDs) possess the same set of function variables. In case of BDDs and ADDs this assumption is without significance, since the variables of skipped levels are always interpreted as *dnc*-variables, where *dnc*-variables do not change the monomial as induced by a respective path and thus do not influence the semantics of the represented function (cf. Sec. 3.2.1, p. 35). However, in the context of *z*-BDDs skipped variables within a variable sequences as induced by a path $p := (n_r, \ldots, n_q, t)$ have different semantics. This semantic depends on the value of the terminal node t and the level of the final non-terminal node n_q:

(1) $\mathtt{value}(t) = 0 \wedge \mathtt{var}(n_q) <_\pi \vec{\mathsf{v}}[n_{\mathcal{V}}]$:
Skipped variables with larger order than the variable labeling node n_q are interpreted as referring to *dnc*-variables. This interpretation arises from the fact that the incoming edges of the fictitious nodes of the skipped levels are re-directed to the 0-child, which is here also its 1-child, namely the terminal *0-node*. For exemplification one may refer to *z*-BDD Z_3 depicted in Fig. 3.5. On the dashed line v_2 and v_3 are skipped, yielding a *dnc*-semantics for them on the resp. path. Consequently the assignments (0,0,0); (0,0,1); (0,1,0); (0,1,1); and (1,0,0); (1,0,1); obtain the function value 0.

(2) all other cases:
Each variable skipped within a path is interpreted as being 0-assigned. Consequently the bit at position i of the assignment as induced by a path p takes the value 0 and thus the derived minterm contains $\neg\vec{v}[i]$. For exemplification one may refer once again to *z*-BDD Z_3 of Fig. 3.5, but this time to the path leading to the terminal *1-node*. Here variable v_3 is skipped, which gives one the assignment (1,1,0) due to the *0-sup.*-reduction rule in case of v_3.

Consequently the function as represented by a ZDD-node and its subgraph can only be deduced correctly, if the set of variables is known. Furthermore also the algorithms for manipulating ZDDs only work properly, if the set of variables the respective ZDDs are defined on are given. This is not problematic as long as one operates within a shared BDD-environment, where each ZDD has the same set of variables. **Consequently within a non-fully shared BDD-environment a ZDD-node alone does not define a boolean function.** I.e. if allocating ZDDs in a shared BDD-environment, where a variable is supposed not to be decisive for a specific ZDD, a case distinction must always be made, where one always checks whether a variable belongs to a given ZDD or not while traversing the respective ZDD. In the following, ZDDs which are defined on a subset of the variables of the entire BDD-environment, will be denoted as *partially shared* ZDDs (*p*ZDDs). *The set of variables, i.e. the set of all variables defined in a BDD-environment is from now on denoted* $\mathcal{V}^G$. Since *z*-BDDs are a special case of ZDDs, also often denoted as 0-1 ZDDs, it is furthermore clear that the concepts and algorithms to be developed in the following are also applicable there.

3.3.1 Definitions

Basically one has two choices for making the graphs of *p*ZDDs a canonical representation of boolean functions:

(1) Inserting *dnc*-nodes at the respective levels, which is the strategy which was applied in Fig. 3.5 for *z*-BDD Z_1 and for the ZDDs depicted in Fig. 3.7. By doing so, one artifically converts the *p*ZDDs into fully shared ZDDs, allowing the use of well known BDD-algorithms [Bry86], adapted to the *0-sup.*-reduction rule (cf. [Bai05]). However, doing so comes at the cost of significantly increasing the memory consumption and thus also time efficiency of the resulting implementation.

(2) One may equip each node n with the set of variables $\mathcal{V}^n$ the function to be represented is defined over.

In the following design alternative two will be applied.

As already indicated in Def. 3.2 (p. 34), together with the canonicity of BDDs, it is clear that within a shared BDD-environment, the nodes representing the same (pseudo-) boolean function are merged. However, for *p*ZDDs this means, two nodes n and m can only be considered as isomorphic *iff* their sub-graphs and their set of variables of the resp. functions are also identical. This leads one to the following definition:

Definition 3.8: Set of function-variables

Let the function $\texttt{varset} : \mathcal{K} \mapsto \mathcal{V}$ be defined, assigning a set of function variables ($\mathcal{V}^n$) to each node n. The elements of $\mathcal{V}^n$ are denoted as *function*-variables. Since the relation π is still defined on the function-variables, the elements of each $\mathcal{V}^n$ can be mapped to a vector $\vec{\mathsf{v}}^n$, so that $\vec{\mathsf{v}}^n[i]$ refers to the i'th variable and the i'th bit position of a resp. assignment.

The above definition forces one to define a set of function-variables for each node. However this also allows to directly deduce the represented function from the graph. E.g. assigning the empty set to the terminal non-*0-node*s yields, that a terminal non-*0-node* n represents the constant $\texttt{value}(n)$-function with $\texttt{value}(n) \neq 0$, rather than the function:

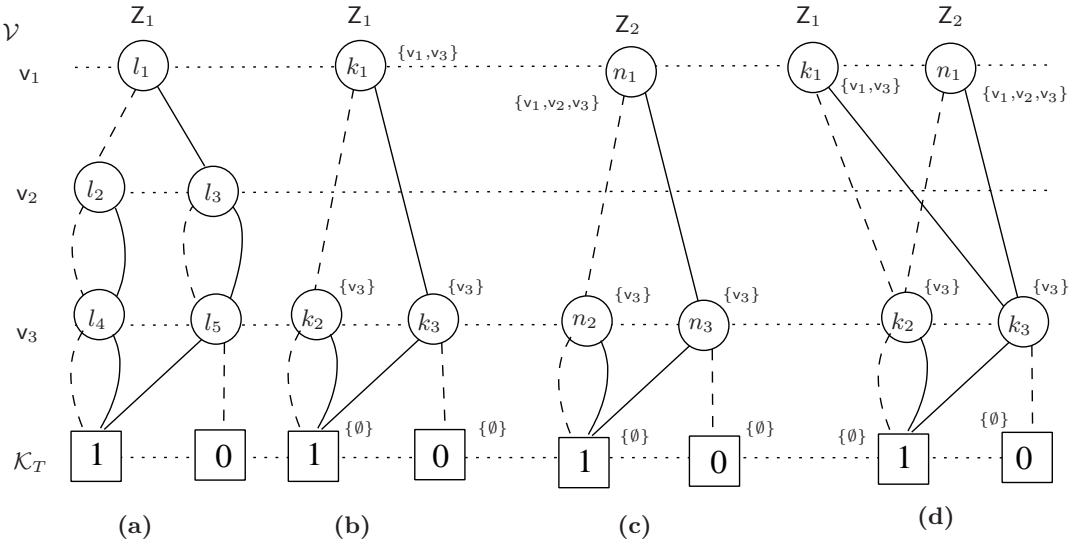


Figure 3.8: *p*ZDD based representation of boolean functions

$$f^n(\vec{b}) := \begin{cases} \texttt{value}(n) \Leftrightarrow \vec{b} = \vec{0} \\ 0 \qquad\qquad otherwise \end{cases}$$

as it would be the case in a shared BDD-environment. Contrary to this, the terminal *0-node* always represents the constant 0-function.

Based on the set of function-variables, one is enabled to re-define the isomorphism-rule for nodes, yielding:

Definition 3.9: Isomorphism of ZDD-nodes

Two ZDD nodes $n, m \in \mathcal{K}$ are isomorphic *iff* they are isomorphic according to Def. 3.2 (p. 34) and their sets of *function*-variables are identical, i.e.:

(1) Non-terminal case: $n, m \in \mathcal{K}_{NT}$

$$n \equiv m \Leftrightarrow \texttt{var}(n) = \texttt{var}(m) \ \wedge \ \texttt{else}(n) = \texttt{else}(m) \ \wedge \ \texttt{then}(n) = \texttt{then}(m) \ \wedge \ \mathcal{V}^n = \mathcal{V}^m$$

(2) Terminal case $n, m \in \mathcal{K}_T$: $n \equiv m \Leftrightarrow \texttt{value}(n) = \texttt{value}(m) \wedge \mathcal{V}^n = \mathcal{V}^m$.

For exemplification one may refer to Fig. 3.8. If the ZDDs of Fig. 3.8.a and b are both interpreted as standard shared ZDDs, i.e. $\mathcal{V}^{\mathsf{Z}_1} = \mathcal{V}^G$, they represent different functions, namely the boolean function $f_{\mathsf{Z}_1} := \neg\mathsf{v}_1 + \mathsf{v}_1\mathsf{v}_3$ in case of Fig. 3.8.a and $f_{\mathsf{Z}_1} := \neg\mathsf{v}_1\neg\mathsf{v}_2 + \mathsf{v}_1\neg\mathsf{v}_2\mathsf{v}_3$ in case of Fig. 3.8.b. However, if the variables v_1 and v_3 are the only function variables for the function represented by node k_1, the graphs of Fig. 3.8.a and b are interpreted as the same function. In contrast the ZDDs of Fig. 3.8.b and 3.8.c have isomorphic graphs, with respect to Def. 3.2 (p. 34). In a shared BDD-environment they consequently would represent the same function and would be stored as a single ZDD, which is truly not what was intended. By defining sets of function-variables for their root nodes, where $\mathcal{V}^{n_1} = \mathcal{V}^G$, and $\mathcal{V}^{k_1} = \{\mathsf{v}_1, \mathsf{v}_3\}$, and interpreting each graph over these sets, the intended different interpretation is achieved. Consequently within a BDD-environment, where each node is intended to represent a unique boolean function, node k_1 and n_1 can not be merged. This is achieved by strictly applying the isomorphism-rule of Def. 3.9. Its application is illustrated in Fig. 3.8.d where Z_2 and Z_3 are stored as multi-rooted ZDD. In contrast to node n_1 and k_1 the nodes n_2 and k_2, as well as n_3 and k_3 can be merged, since they have isomorphic sub-graphs and identical sets of function-variables. Embracing the above discussion, we define *p*ZDDs now

Algorithm 3.1 Function for allocating unique *p*ZDD nodes only

```
getUniquepZDDNode(v, t, e, V^i)
(0)  IF t = 0-node THEN ;
(1)    IF e ∉ K_T THEN ;
(2)      RETURN getUniquepZDDNode(v, then(e), else(e), V^i);
(3)    ELSE
(4)      RETURN getTerminalpZDDNode(V^i, 1);
(5)  ELSE
(6)  K^v_NT := {n ∈ K_NT | var(n) = v};
(7)  IF ∃n ∈ K^v_NT : then(n) = t && else(n) = e && V^n = V^i THEN
(8)    RETURN n;
(9)  END
(10) n := getNode(v, t, e, V^i);
(11) K_NT ⟵ n;
(12) RETURN n;
```

formally as follows:

Definition 3.10: Multi-terminal partially shared zero-suppressed BDD

A partially shared zero-suppressed multi-terminal BDD (*p*ZDD) $\mathsf{Z} < \mathcal{V}^{\mathsf{Z}}, \pi, \mathbb{D} >$ is a ZDD $\mathsf{Z} := \{\mathcal{K}, \mathtt{var}, \mathtt{then}, \mathtt{else}, \mathtt{varset}\}$, where

(1) each node $u \in \mathcal{K}$ is equipped with a set of *function-variables* $\mathtt{varset}(u) := \mathcal{V}^u$ (Def. 3.8), and

(2) $\nexists u, v \in \mathcal{K}^{\mathsf{Z}} : u \neq v \wedge u \equiv v$ according to Def. 3.9 holds.

The new rule for node isomorphism can once again be encapsulated in the function `getUniquepZDDNode` for allocating unique *p*ZDD-nodes. As input parameters this algorithm takes the variable the allocated node needs to be equipped with (`v`), its `then` and `else`-child, and its set of function variables (Algo. 3.1, p. 43).

An example for the *p*ZDD based representation of pseudo boolean functions is given in Fig. 3.9 (p. 44). The function tables are given in Fig. 3.9.i and 3.9.ii. The drawn *p*ZDDs are intended to represent the same pseudo boolean functions as illustrated in Fig. 3.7. However, in contrast to Fig. 3.7, one is now enabled to omit the *dnc*-nodes referring to non-function variables. Consequently the mapping of bit position b_i to a variable $\mathtt{v}_j$ as shown in table (i) and (ii) is also adapted.

3.3.2 Canonicity of *p*ZDDs

Now we will prove that *p*ZDDs are a (weak) canonical form of representing (pseudo-) boolean functions.

Lemma 3.1: *Each p*ZDD*-node n with its set of function variables* $\mathcal{V}^n$ *represents a (pseudo-) boolean function.* $f^n : \mathbb{B}^{|\mathcal{V}^n|} \rightarrow \mathbb{D}$.

Lemma 3.1 directly arises from Def. 3.10, in combination with Eq. 3.1- 3.3 and the respective version of function `Satisfy` (Algo. B.3, p. 164)

Theorem 3.2: *Each (pseudo-) boolean function* $f : \mathbb{B}^{n_\mathcal{V}} \rightarrow \mathbb{D}$*, with a fixed set of function variables (*$\mathcal{V}$*) and a fixed total ordering relation* π *on* $\mathcal{V}$*, is represented by a unique p*ZDD*-graph rooted in a p*ZDD*-node n.*

Proof: The proof directly arises from algorithm `getUniquepZDDNode` (Algo. 3.1), which is based on the Shannon expansion (Def. A.1, p. 162) and the definition of *p*ZDD-nodes. As a consequence, this does not give any degree of freedom when constructing *p*ZDD based

(i) Function f_{B}

$\mathsf{v}_3\mathsf{v}_4\mathsf{v}_5\mathsf{v}_6$	$f_B(\vec{b})$
$b_1b_2b_3b_4$	
1 0 0 1	δ
1 1 1 0	
0 1 0 1	γ
0 1 0 0	
0 0 1 0	μ

(ii) Function f_{A}

$\mathsf{v}_1\mathsf{v}_2$	$f_A(\vec{b})$
b_1b_2	
1 1	1
0 0	1

(iii) pZDDs for f_{A} and f_{B}

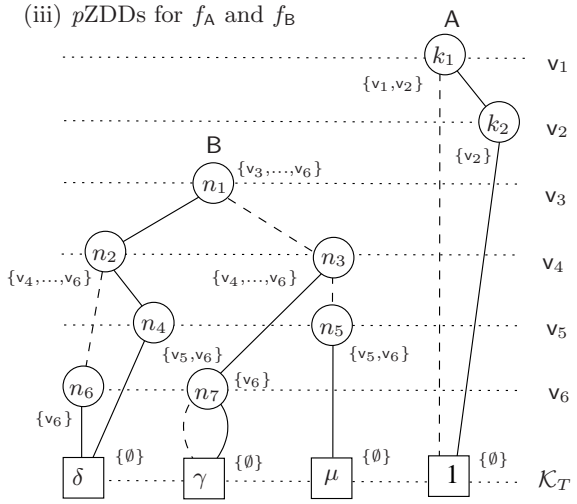


Figure 3.9: Function tables and pZDD based representation of pseudo-boolean function

representations of pseudo-boolean functions. The proof will therefore be carried out by induction.

(1) Start of induction with $n_{\mathcal{V}} = 0$:
Lemma 3.1 gives that Theorem 3.2 is true for $n \in \mathcal{K}_T$: I.e. the constant 0-ary functions for $\texttt{value}(n) \in \mathbb{D}$ are represented by $n \in \mathcal{K}_T$, where $\mathcal{V}^n = \emptyset \Leftrightarrow n_{\mathcal{V}} = 0$.

(2) Proposition of induction:
For each pseudo boolean $n_{\mathcal{V}}$-ary function there exists a unique pZDD-graph rooted in a pZDD-node n.

(3) Induction step from $n_{\mathcal{V}}$ to $n_{\mathcal{V}} + 1$:
Any pseudo-boolean $(n_{\mathcal{V}} + 1)$-ary function f^n can be expanded as: $f^n := \mathsf{v}_j f_1^n + \neg\mathsf{v}_j f_0^n$ (cf. Def. A.1, p. 162), where each of the co-factors represents a (pseudo-) boolean $n_{\mathcal{V}}$-ary function. The induction's proposition yields that each of these $n_{\mathcal{V}}$-ary function can be represented by the unique pZDD-nodes l and k. The pZDD-node $n := \texttt{getUnique}\textit{p}\texttt{ZDDNode}(\texttt{var}(n), l, k, \mathcal{V}^n)$ is constructed in such a way, that the unique nodes l and k are its children nodes and $\mathcal{V}^n := \{\texttt{var}(n)\} \cup \mathcal{V}^l \cup \mathcal{V}^k$ (cf. Algo. 3.1, p. 43). Thus one obtains a unique pZDD-node n representing the $(n_{\mathcal{V}} + 1)$-ary function f^n. –It is important to note that in case n is a *0-sup.* node, `getUnique`*p*`ZDDNode` will return not the as argument passed `else`-child, but a node of the respective level. which is also equipped with the set $\mathcal{V}^n$ as its set of function variables.

■

Theorem 3.3: *For a fixed order defined on the function variables, a* pZDD *is a (weak) canonical representation of a pseudo-boolean function.*

Proof: Theorem 3.2 gives that the mapping of a pseudo-boolean function f to a pZDD-node is injective. Lemma 3.1 gives, that each pZDD-node represents a pseudo boolean function f (surjectivity). Consequently the mapping of pseudo boolean functions to a pZDD-node is bijective. ■

For making pZDDs a strong canonical representation, one must simply eliminate *dnc*-nodes within the graphs and remove the associated variable from the set of function variables. This can be achieved by extending the node allocating function `getUnique`*p*`ZDDNode` accordingly.

However, in such a setting, one would either need to equip each 0 or 1-edge with its own set of decisive variables or choose the set of function variables for each node in such a way that only the variables of *0-sup.*-nodes from larger order are recorded, for exemplification one may refer to Fig. 3.9.iii and the graph rooted in node n_3. Since the maintenance of canonicity in case of partially shared ZDDs may increase the number of DD-nodes anyway, no matter if weak or strong canonicity is to be achieved, we will apply a different strategy, when it comes to implementing them.

3.4 Operations on *p*ZDDs

For efficiently manipulating BDDs and their extensions, many recursive algorithms are known. In the following we will introduce only those algorithms which are from concern for this work. We will not go into details, concerning their BDD based variants, since they are well known from standard literature (e.g. [Bry86, Min93, DB98, MT98, SF96, Bai05]).

3.4.1 Preliminaries

It is possible to implement *p*ZDDs with different set of function variables in a common shared BDD-environment, where only node-isomorphism according to Def. 3.2 is realized! Such a strategy has the main advantage that one (a) does not need to store the sets of function variables for each node and (b) increases the sharing among the *p*ZDD-graphs. But then a *p*ZDD- node does not anymore represent necessarily a unique (pseudo-) boolean function. As a consequence implementation is intricate, namely if testing two *p*ZDD-nodes for being equal, node-isomorphism of Def. 3.2 is not sufficient, the set of function variables must also be compared! But within a standard BDD-environment these sets are in principle not obtainable. For solving this dilemma one may proceed as follows: When operating on a *p*ZDD Z one simply passes its set of function variables ($\mathcal{V}^{\mathsf{Z}}$) as additional argument to the respective algorithm. These sets can be efficiently stored as a *z*-BDD *cube* representing the boolean function $\bigwedge_{\mathcal{V}^{\mathsf{Z}}} \mathsf{v}_j$, also commonly denoted as cube set. A storage of these sets as cube sets not only yields space efficiency but also allows an efficient iteration on them, while recursing towards the terminal nodes of the *p*ZDDs. As a consequence of such a strategy, one significantly decreases time and space complexity of *p*ZDDs. However, one the other hand the recursion of the *p*ZDD-manipulating algorithms may not terminate their recursion as soon as possible, since each (redundant) *dnc*-node within the graphs may impose two additional recursive calls, *which in case of strong canonicity could be prevented.* For exemplification one may refer once again to Fig. 3.8.a, and consider that each node represents a boolean function. In case of weak canonicity, the computation $(l_5 \wedge l_3)$ must be carried out, even one knows that the result is l_3, since l_5 is a non-decisive variable. But nevertheless in the next step the recursion can be terminated, since $(l_3 \wedge l_3)$ is alway known to be l_3, due to the fact that both nodes are identical.

In the following we assume an DD-environment, where nodes are not equipped with sets of function variables. This allows us to replace function `getUniquepZDDNode` with the less complex function `getUniqueZDDNode`(v_i, e, t), for allocating ZDD-nodes (cf. Sec. 3.2.2). For simplifying the algorithms to be developed in the following, we also re-define the previously introduced function `var`, so that `var` : $\mathcal{K} \mapsto \{\mathcal{V} \cup t\}$, where

$$\texttt{var}(n_i) := \begin{cases} \mathsf{v} \in \mathcal{V} & \textit{iff } n_i \in \mathcal{K}_{NT} \\ \mathsf{t} & \textit{iff } n_i \in \mathcal{K}_T \end{cases}$$

This gives a variable-labeling of each terminal node with the special variable $\mathsf{t} \notin \mathcal{V}$. Consequently the order π must also be extended accordingly:

$$\pi \subseteq \{\mathcal{V} \cup \mathsf{t}\} \times \{\mathcal{V} \cup \mathsf{t}\},$$

where t is defined to have the highest order with respect to the elements of $\mathcal{V}$.

3.4.2 Applying binary operators to *p*ZDDs

A binary operator `op` can be applied directly to BDDs and its derivatives by the help of Bryant's recursive `Apply`-algorithm [Bry86] or variants thereof. The `Apply`-algorithm and its variants constructs a new BDD B, representing the function $f_{\mathsf{B}} := f_{\mathsf{B}_1}\ \mathtt{op}\ f_{\mathsf{B}_2}$. I.e. in the following $\mathsf{B} := \mathsf{B}_1 \mathtt{op}\ \mathsf{B}_2$ is always carried out by the help of the *respective* version of the `Apply`-algorithm, where the syntax of the initial call is given by $\mathtt{Apply}(\mathtt{op}, \mathtt{getRoot}(\mathsf{B}_1), \mathtt{getRoot}(\mathsf{B}_2))$. The called version of `Apply` depends on the circumstances, whether one operates on DDs where the *dnc-* or the *0-sup.* has been applied.

In case of DDs, where reduction rules have been applied, one traverses in general not jointly on both DDs. Consequently one may reach the terminal nodes of one of them earlier, where the partner DD still needs to be traversed. In some of these cases, it is possible to terminate the recursion of the traversing algorithms, which of course depends on the applied binary operator and the employed DD type, e.g. for a dnc-*free* BDD the case $0 \wedge \mathsf{B}$ will always give the constant 0-function. I.e. with respect to universality, one can employ operator specific functions, which steer the recursion of the traversing algorithms. I.e. the traversing algorithm calls the respective `op`-function, rather than testing the terminal condition for the respective binary operator itself. The called `op`-function returns then either a node, representing the result of $f_n\, \mathtt{op} f_m$, where n and m are the as argument passed nodes, or it returns the empty node ϵ. In this latter case the recursion of the traversing algorithm must be continued, since the terminal condition for terminating the recursion is not satisfied. Since the implemented functions not only steer the recursion depth of the *p*ZDD-traversing algorithm, but also implement the functionality of the respective boolean or arithmetic operator, mostly on the level of terminal nodes only, we denote them as operator functions (`op`-functions). They will be discussed in greater detail after the generic *p*`ZApply`-algorithm has been introduced.

Before introducing the *p*`ZApply`, we will first execute the Shannon expansion for all different cases as appearing when applying a binary operator `op` to (pseudo-)boolean functions represented by *p*ZDDs. Doing so will determine the recursive branching as to be implemented into a respective algorithm.

Applying the Shannon expansion

Let pseudo-boolean function g be represented by *p*ZDD A and function h by *p*ZDD B. If one wishes to compute $f := g\ \mathtt{op}\ h$, i.e. apply a binary operator to the *p*ZDDs A and B, the respective call to the *p*`ZApply`-algorithm may look like $p\mathtt{ZApply}(\mathtt{op}, n, \mathcal{V}^n, m, \mathcal{V}^m, cube)$, where

(1) n and m are the root nodes of A and B.

(2) $\mathcal{V}^n$ and $\mathcal{V}^m$ are their sets of function variables.

(3) The *z*-BDD cube represents f's function variables, encoded as cube set, so that *cube* can serve as stack to the variables to be handled by the different recursions.

Let the nodes n and m be labeled with the variables v_n and v_m. For executing the Shannon expansion, it is now only necessary to consider an arbitrary variable $\mathsf{v}_c \in \mathcal{V}^G$ and to cover all possible settings. I.e. depending on the variable labels v_c, v_n and v_m the recursion of an *p*ZDD based algorithm has to implement the following expansions:

Variable v_c *is a function variable for both p*ZDD*s*

Since $\mathsf{v}_c \in \mathcal{V}^n \cap \mathcal{V}^m$ holds, the current variable is a function variable for both *p*ZDDs as rooted in n and m. However, it might be skipped within one, both or none of the graphs, which yields the following case distinctions:

(1) No skipping of a variable appeared, i.e. $\mathsf{v}_c = \mathsf{v}_n = \mathsf{v}_m$, and consequently the nodes n and m are both allocated at the level assigned to the current variable. This situation is covered by the following expansion:

$$\begin{aligned} f &:= h \,\mathtt{op}\, g \\ &= (\mathsf{v}_c h_1 + \bar{\mathsf{v}}_c h_0) \,\mathtt{op}\, (\mathsf{v}_c g_1 + \bar{\mathsf{v}}_c g_0) \\ &= \mathsf{v}_c\mathsf{v}_c(h_1 \,\mathtt{op}\, g_1) + \bar{\mathsf{v}}_c\mathsf{v}_c(h_0 \,\mathtt{op}\, g_1) + \mathsf{v}_c\bar{\mathsf{v}}_c(h_1 \,\mathtt{op}\, g_0) + \bar{\mathsf{v}}_c\bar{\mathsf{v}}_c(h_0 \,\mathtt{op}\, g_0) \\ &= \mathsf{v}_c(h_1 \,\mathtt{op}\, g_1) + \bar{\mathsf{v}}_c(h_0 \,\mathtt{op}\, g_0) \end{aligned} \tag{3.6}$$

As one can see, for constructing the pZDD Z representing function f, one simply recurses into the `then`-branch by executing $h_1 \mathtt{op} g_1$ and into the `else`-branch by executing $h_0 \mathtt{op} g_0$.

(2) A skipping of a variable within one of the pZDDs appeared, yielding the following case distinctions:

(2.a) $\mathsf{v}_n \,{}_\pi{>}\, \mathsf{v}_c \wedge \mathsf{v}_m = \mathsf{v}_c$, i.e. node m is labeled with v_c, where node n is labeled with a variable of higher order. Consequently v_c is a *0-sup.*-variable within the graph rooted in n. This situation is resolved by the following expansion:

$$\begin{aligned} f &:= h \,\mathtt{op}\, g \\ &= (\mathsf{v}_c h_1 + \bar{\mathsf{v}}_c h_0) \,\mathtt{op}\, (\mathsf{v}_c g_1 + \bar{\mathsf{v}}_c g_0) \\ &= \mathsf{v}_c\mathsf{v}_c(h_1 \,\mathtt{op}\, g_1) + \bar{\mathsf{v}}_c\mathsf{v}_c(h_0 \,\mathtt{op}\, g_1) + \mathsf{v}_c\bar{\mathsf{v}}_c(h_1 \,\mathtt{op}\, g_0) + \bar{\mathsf{v}}_c\bar{\mathsf{v}}_c(h_0 \,\mathtt{op}\, g_0) \\ &= \mathsf{v}_c(h_1 \,\mathtt{op}\, 0) + \bar{\mathsf{v}}_c(h_0 \,\mathtt{op}\, g_0) \end{aligned} \tag{3.7}$$

While expanding function h as usual, the expansion of function g follows the *0-sup.*-rule. I.e. for the `else`-branch g and g_0 are represented by the same node, namely node n. Within the `then`-branch function g_1 is evaluated to 0. Thus one needs to evaluate here $h_1 \,\mathtt{op}\, 0$.

(2.b) $\mathsf{v}_m \,{}_\pi{>}\, \mathsf{v}_c \wedge\ \mathsf{v}_n = \mathsf{v}_c$, this case is symmetric to the one above.

(3) The current variable is skipped within both pZDDs i.e. $\mathsf{v}_n, \mathsf{v}_m \,{}_\pi{>} \mathsf{v}_c$. Since v_c is a function variable for both pZDDs, it must be *0-sup.* for both pZDDs, which is expressed by the following expansion:

$$\begin{aligned} f &:= h \,\mathtt{op}\, g \\ &= (\mathsf{v}_c h_1 + \bar{\mathsf{v}}_c h_0) \,\mathtt{op}\, (\mathsf{v}_c g_1 + \bar{\mathsf{v}}_c g_0) \\ &= \mathsf{v}_c\mathsf{v}_c(h_1 \,\mathtt{op}\, g_1) + \bar{\mathsf{v}}_c\mathsf{v}_c(h_0 \,\mathtt{op}\, g_1) + \mathsf{v}_c\bar{\mathsf{v}}_c(h_1 \,\mathtt{op}\, g_0) + \bar{\mathsf{v}}_c\bar{\mathsf{v}}_c(h_0 \,\mathtt{op}\, g_0) \\ &= \mathsf{v}_c(0 \,\mathtt{op}\, 0) + \bar{\mathsf{v}}_c(h_0 \,\mathtt{op}\, g_0) \end{aligned} \tag{3.8}$$

Since v_c is a *0-sup.*-variable for function h and g, the operator function, as called within the next recursive step, can terminate the `then`-branch recursion, as long as the called `op`-function is a *0-maintaining* function (cf. Appendix A, p. 161). Otherwise one must recurse with the remaining variables. In case of the `else`-branch recursion one resumes with $h_0 \,\mathtt{op}\, g_0$.

Variable v_c is a function variable for only one of the pZDDs

Now the case of v_c being a non-function variable for one of the pZDDs rooted in n, m resp. is covered ($\mathsf{v}_c \notin \mathcal{V}^n \cap \mathcal{V}^m$ but $\mathsf{v}_c \in \mathcal{V}^n \cup \mathcal{V}^m$), where the following situations may appear:

(1) The level referring to variable v_c is skipped within the graph rooted in n, i.e. v_c is not a function variable for this pZDD e.g. the one representing function g ($\mathsf{v}_c = \mathsf{v}_m \wedge \mathsf{v}_c <_\pi \mathsf{v}_n$). This situation can be handled by the following expansion:

$$\begin{aligned} f &:= h \,\mathtt{op}\, g \\ &= (\mathsf{v}_c h_1 + \bar{\mathsf{v}}_c h_0) \,\mathtt{op}\, (\mathsf{v}_c g_1 + \bar{\mathsf{v}}_c g_0) \\ &= \mathsf{v}_c\mathsf{v}_c(h_1 \,\mathtt{op}\, g) + \bar{\mathsf{v}}_c\mathsf{v}_c(h_0 \,\mathtt{op}\, g) + \mathsf{v}_c\bar{\mathsf{v}}_c(h_1 \,\mathtt{op}\, g_0) + \bar{\mathsf{v}}_c\bar{\mathsf{v}}_c(h_0 \,\mathtt{op}\, g_0) \\ &= \mathsf{v}_c(h_1 \,\mathtt{op}\, g_1) + \bar{\mathsf{v}}_c(h_0 \,\mathtt{op}\, g_0) \text{ where } g = g_i \text{ for } i \in \{0,1\} \\ &= \mathsf{v}_c(h_1 \,\mathtt{op}\, g) + \bar{\mathsf{v}}_c(h_0 \,\mathtt{op}\, g) \end{aligned} \tag{3.9}$$

As one can see while expanding function h, nothing needs to be done for function g, since the variable v_c is not a function variable for this function.

(2) For $\mathsf{v}_c = \mathsf{v}_n \wedge \mathsf{v}_c <_\pi \mathsf{v}_m$ one obtains the symmetric case to the above one.

(3) The level referring to variable v_c is skipped within the graph rooted in n, m respectively. However, v_c is *0-sup.* for the former *p*ZDD and not a function variable of the latter ($\mathsf{v}_n, \mathsf{v}_m <_\pi \mathsf{v}_c \wedge \mathsf{v}_c \notin \mathcal{V}^m$), so that the following expansion applies:

$$\begin{aligned} f &:= h \,\mathtt{op}\, g \\ &= (\mathsf{v}_c h_1 + \bar{\mathsf{v}}_c h_0) \,\mathtt{op}\, (\mathsf{v}_c g_1 + \bar{\mathsf{v}}_c g_0) \\ &= \mathsf{v}_c \mathsf{v}_c (h_1 \,\mathtt{op}\, g_1) + \bar{\mathsf{v}}_c \mathsf{v}_c (h_0 \,\mathtt{op}\, g_1) + \mathsf{v}_c \bar{\mathsf{v}}_c (h_1 \,\mathtt{op}\, g_0) + \bar{\mathsf{v}}_c \bar{\mathsf{v}}_c (h_0 \,\mathtt{op}\, g_0) \\ &= \mathsf{v}_c (h \,\mathtt{op}\, 0) + \bar{\mathsf{v}}_c (h \,\mathtt{op}\, g) \end{aligned} \tag{3.10}$$

Since v_c is a *dnc*-variable for function h and a *0-sup.* one for function g, the `then`-branch recursion simply takes h and 0 as its argument. In case of the `else`-branch recursion one resumes with $h \,\mathtt{op}\, g$.

(4) In case $\mathsf{v}_n, \mathsf{v}_m <_\pi \mathsf{v}_c \wedge \mathsf{v}_c \notin \mathcal{V}^n$ one obtains the symmetric case to the above one.

Remark 3.4: Due to the above results a *p*ZDD based algorithm, facing any of the above four cases is forced to consider each variable as contained in $\mathcal{V}^n \cup \mathcal{V}^m$. Consequently in contrast to the known variants of the `Apply`-algorithm, it is not sufficient for the algorithm to stop only at levels, where actually nodes within the graphs of f^n and f^m are appearing.

*Variable v_c is non-decisive for both p*ZDD*s*
In case $\mathsf{v}^c \notin \mathcal{V}^n \cup \mathcal{V}^m$ it appears that variable v_c is skipped in both recursions, since v_c is not a function variable for both *p*ZDDs. Consequently the *dnc*-semantics is applicable for both *p*ZDDs, yielding the following expansion:

$$\begin{aligned} f &:= h \,\mathtt{op}\, g \\ &= (\mathsf{v}_c h_1 + \bar{\mathsf{v}}_c h_0) \,\mathtt{op}\, (\mathsf{v}_c g_1 + \bar{\mathsf{v}}_c g_0) \\ &= \mathsf{v}_c \mathsf{v}_c (h_1 \,\mathtt{op}\, g) + \bar{\mathsf{v}}_c \mathsf{v}_c (h_0 \,\mathtt{op}\, g) + \mathsf{v}_c \bar{\mathsf{v}}_j (h_1 \,\mathtt{op}\, g_0) + \bar{\mathsf{v}}_c \bar{\mathsf{v}}_c (h_0 \,\mathtt{op}\, g_0) \\ &= \mathsf{v}_c (h_1 \,\mathtt{op}\, g_1) + \bar{\mathsf{v}}_c (h_0 \,\mathtt{op}\, g_0) \text{ where } h = h_i \text{ and } g = g_i \text{ for } i \in \{0,1\} \\ &= h \,\mathtt{op}\, g \\ &= f \end{aligned} \tag{3.11}$$

Since v_c is non-decisive for functions g and h, nothing needs to be done, one simply resumes with the recursion for expanding h and g, in order to compute $f := h \,\mathtt{op}\, g$.

Remark 3.5: The above result yields the nice feature that for variables $\mathsf{v}_c \in \mathcal{V}^G \setminus (\mathcal{V}^n \cup \mathcal{V}^m)$ nothing needs to be done. Consequently the *z*-BDD *cube* must not be defined over $\mathcal{V}^G$, but over $\mathcal{V}^n \cup \mathcal{V}^m$.

The generic *p*`ZApply`-algorithm
The pseudo-code of the generic *p*`ZApply`-algorithm is specified as Algo. 3.2. As input parameters the algorithm takes the binary operator to be executed, the pair of nodes the functions to be combined are rooted in (n, m), their set of function variables ($\mathcal{V}^n$, $\mathcal{V}^m$) and the cube-set over the variables the resulting function is defined over (*cube*). Alternatively one may employ $\mathsf{v}_c := min(\mathcal{V}^n, \mathcal{V}^m)$ within the algorithm, instead of recursing on *cube*. But for efficiently implementing the sets, the employment of a *z*-BDD *cube* is recommended and we will therefore employ it in the following algorithms.
At the first call n and m are the root-nodes of the respective *p*ZDDs A and B. The variable sets $\mathcal{V}^n$ and $\mathcal{V}^m$ are their individual set of function variables, and the *z*-BDD cube encodes the cube-set over $\mathcal{V}^{\mathsf{A}} \cup \mathcal{V}^{\mathsf{B}}$.

In line 1 and 2 the terminal condition with the help of the respective operator function

Algorithm 3.2 The generic p`ZApply`-algorithm

p`ZApply`$(op, n, \mathcal{V}^n, m, \mathcal{V}^m, cube)$
(0) *node* res, e, t;
(1) $res := op(n, \mathcal{V}^n, m, \mathcal{V}^m)$;
(2) IF $res \neq \epsilon$ THEN RETURN res;

/* *Check* `op`*-cache if result is already known* */
(3) $res =$ `cacheLookup`$(op, n, \mathcal{V}^n, m, \mathcal{V}^m)$;
(4) IF $res \neq \epsilon$ THEN RETURN res;

/* *Remove variables from set* */
(5) $\mathcal{V}^n_{new} := \mathcal{V}^n \setminus$ `var`$(cube)$
(6) $\mathcal{V}^m_{new} := \mathcal{V}^m \setminus$ `var`$(cube)$

/* *(A) No level is skipped* */
(7) IF `var`(n), `var`$(m) =_\pi$ `var`$(cube)$ THEN
(8) $e :=$ p`ZApply`$(op,$ `else`$(n), \mathcal{V}^n_{new},$ `else`$(m), \mathcal{V}^m_{new},$ `then`$(cube))$;
(9) $t :=$ p`ZApply`$(op,$ `then`$(n), \mathcal{V}^n_{new},$ `then`$(m), \mathcal{V}^m_{new},$ `then`$(cube))$;

/* *(B) Skipped a level exclusively within one of the* p*ZDD* */
(10) ELSE IF `var`$(n) =_\pi$ `var`$(cube)$ THEN
(11) $e :=$ p`ZApply`$(op,$ `else`$(n), \mathcal{V}^n_{new}, m, \mathcal{V}^m_{new},$ `then`$(cube))$;
(12) IF `var`$(cube) \in \mathcal{V}^m$ THEN
(13) $t :=$ p`ZApply`$(op,$ `then`$(n), \mathcal{V}^n_{new},$ *0-node*$, \emptyset,$ `then`$(cube))$;
(14) ELSE
(15) $t :=$ p`ZApply`$(op,$ `then`$(n), \mathcal{V}^n_{new}, m, \mathcal{V}^m_{new},$ `then`$(cube))$;
(16) ELSE IF `var`$(m) =_\pi$ `var`$(cube)$ THEN
(17) $e :=$ p`ZApply`$(op, n, \mathcal{V}^n_{new},$ `else`$(m), \mathcal{V}^m_{new},$ `then`$(cube))$;
(18) IF `var`$(cube) \in \mathcal{V}^n$ THEN
(19) $t :=$ p`ZApply`$(op,$ *0-node*$, \emptyset,$ `then`$(m), \mathcal{V}^m_{new},$ `then`$(cube))$
(20) ELSE
(21) $t :=$ p`ZApply`$(op, n, \mathcal{V}^n_{new},$ `then`$(m), \mathcal{V}^m_{new},$ `then`$(cube))$;

/* *(C) Skipped a level within both* p*ZDDs* */
(22) ELSE
(23) $e :=$ p`ZApply`$(op, n, \mathcal{V}^n_{new}, m, \mathcal{V}^m_{new},$ `then`$(cube))$;
(24) IF `var`$(cube) \in \mathcal{V}^n$ && `var`$(cube) \in \mathcal{V}^m$
(25) $t :=$ p`ZApply`$(op,$ *0-node*$, \emptyset,$ *0-node*$, \emptyset,$ `then`$(cube))$;
(26) ELSE IF `var`$(cube) \in \mathcal{V}^n$
(27) $t :=$ p`ZApply`$(op,$ *0-node*$, \emptyset, m, \mathcal{V}^m_{new},$ `then`$(cube))$;
(28) ELSE IF `var`$(cube) \in \mathcal{V}^m$
(29) $t :=$ p`ZApply`$(op, n, \mathcal{V}^n_{new},$ *0-node*$, \emptyset,$ `then`$(cube))$;

/* *Allocate new node, respecting (*p*ZDD) isomorphism and 0-sup. rule* */
(30) $res :=$ `getUniqueZDDNode`$(\text{`var`}(cube), t, e\})$;

/* *Insert result into* `op`*-cache and terminate recursion* */
(31) `cacheInsert`$(\text{`op`}, n, \mathcal{V}^n, m, \mathcal{V}^m, res)$;
(32) RETURN res;

is tested. If this is not successful, one checks the `op`-function specific computed table (`op`-cache), if the result is already known from a previous recursion (line 3-4). As one may note, the sets $\mathcal{V}^n$ and $\mathcal{V}^m$ must also be considered, since the set of function variables are not stored within the ZDD-nodes themselves. In case the lookup is not successful, the recursion must be entered, but before doing so the pseudo-code of lines 5 and 6 prepares the new sets of function variables as required in the next recursion.

The different recursions are covered by the pseudo-code of lines 7-29:

- Line 7-9 handles the ordinary branching in case no-skipping of variables appeared within the traversed graphs.

Algorithm 3.3 *p*ZDD `op`-functions for boolean operators

`ZAnd`$(n, \mathcal{V}^n, m, \mathcal{V}^m)$
(0) *node* $res := \epsilon$;
(1) IF $(n = m$ && $\mathcal{V}^n = \mathcal{V}^m)$
(2) THEN $res := n$;
(3) ELSE IF $(n =$ *0-node* $\|$
$m =$ *0-node*$)$
(4) THEN $res :=$ *0-node*;
(5) ELSE IF $(n =$ *1-node* &&
$m =$ *1-node*$)$
(6) THEN $res :=$ *1-node*;
(7) RETURN res;

(a) `op`-function for `op` $= \wedge$

`ZOr`$(n, \mathcal{V}^n, m, \mathcal{V}^m)$
(0) *node* $res := \epsilon$;
(1) IF $(n = m$ && $\mathcal{V}^n = \mathcal{V}^m)$
(2) THEN $res := n$;
(3) ELSE IF $(n =$ *0-node*$)$
(4) THEN $res := m$;
(5) ELSE IF $(m =$ *0-node*$)$
(6) THEN $res := n$;
(7) RETURN res;

(b) `op`-function for `op` $= \vee$

`ZSetMinus`$(n, \mathcal{V}^n, m, \mathcal{V}^m)$
(0) *node* $res := \epsilon$;
(1) IF $(n = m$ && $\mathcal{V}^n = \mathcal{V}^m)$
(2) THEN $res :=$ *0-node*;
(3) ELSE IF $(m =$ *0-node*$)$
(4) THEN $res := n$;
(5) ELSE IF $(n =$ *0-node*$)$
(6) THEN $res :=$ *0-node*;
(7) RETURN res;

(c) `op`-function for `op` $= \setminus$

- The code of lines 10-21 covers the case that the current variable of *cube* is a skipped variable exclusively in one of the graphs. In such a case one executes at first line 11 or 17, for entering the `else`-branch of the recursion. Concerning the `then`-branch, the behavior is more complex. It depends on the circumstances, whether the variable `var`(*cube*) belongs to the set of function-variables of the respective *p*ZDD or it does not. I.e. one either interprets `var`(*cube*) as *0-sup.* - or as a non-decisive variable within the respective graph. In case `var`(*cube*) is considered as being *0-sup.*, line 13 or alternatively line 19 is executed. In case `var`(*cube*) is considered as being non-decisive, one executes line 15 or alternatively line 21.

- Lines 22-29 cover the case that the variable `var`(*cube*) is skipped within both graphs. For the `else`-branch, the current pair of nodes (n and m) is the pair of children nodes, since the 0-children of the fictitious nodes being skipped are the current node n and m themselves (line 23). Concerning the `then`-branch the following cases must be covered: (a) the variable is a function variable for both graphs: here the standard *0-sup.*-branching rules apply, which means that in both cases the *0-node* is the `then`-child to be recursed on (line 25); (b) the variable is a non-function variable for one of the graphs and *0-sup.* for the other. Here the branching rules follows a *dnc*-rule in one case and a *0-sup.*-rule in the other case, which means that in the *dnc*-case one does not traverse any further, where in the *0-sup.*-case a traversal to the *0-node* follows (line 27 and 29).

Finally, when returning from the recursion, one allocates a new *p*ZDD node, representing f^n `op` f^m (line 30). This result is then inserted into the `op`-cache, where also the respective sets of function variables must be provided. Now the algorithm can terminate by passing the result of line 30 as its return value to the calling function.

The `op`-functions

The *p*`ZApply`-algorithm works for all binary boolean and binary arithmetic operators (`op` $\in \{\vee, \wedge, \setminus \ldots\}$ and `op` $\in \{\cdot, +, \ldots\}$), given that a respective `op`-function is provided and no matter if it is *0-maintaining* (cf. Appendix A, p. 161) or not. In case the `op`-function returns a non-empty node the resp. calling *p*`ZApply` terminates the recursion by returning the received node as result. In the following we will discuss now the different `op`-functions each implementing a different binary operator. In the following paragraph we will only discuss the `op`-functions, which are of concern for this work. The `op`-functions for other binary operators can be implemented analogously.

`op`-*functions implementing boolean operators*

(1) In case `op` $= \wedge$ the *p*`ZApply`-algorithm is steered by the `op`-function of Algo. 3.3.a. In case of node-equality the recursion can only be terminated, if also the set of function-variables matches (line 1-2). Due to the different semantic of skipped variables, –one assumes a *dnc-semantics* for the remaining variables in case of reaching a terminal *0-node*, where

Algorithm 3.4 *p*ZDD op-functions for arithmetic operators

`ZTimes`$(n, \mathcal{V}^n, m, \mathcal{V}^m)$
(0) *node* $res := \epsilon$;
(1) IF $(n =$ *0-node* $\| \; m =$ *0-node*$)$
(2) THEN $res :=$ *0-node*;
(3) ELSE IF $(n, m \in \mathcal{K}_T)$ THEN
(4) $v :=$ `value`$(n) \cdot$ `value`(m);
(5) $res :=$ `getTerminalNode`(v);
(6) RETURN res;

(a) op-function for `op` $= \cdot$

`ZDiv`$(n, \mathcal{V}^n, m, \mathcal{V}^m)$
(0) *node* $res := \epsilon$;
(1) IF $(m =$ *0-node*$)$ THEN
(2) `error_exit`
(3) IF $(n =$ *0-node*$)$
(4) THEN $res :=$ *0-node*;
(5) ELSE IF $(n = m \;\&\&\; \mathcal{V}^n = \mathcal{V}^m)$
(6) THEN $res :=$ *1-node*
(7) ELSE IF $(n, m \in \mathcal{K}_T)$ THEN
(8) $v :=$ `value`(n)/`value`(m);
(9) $res :=$ `getTerminalNode`(v);
(10) RETURN res;

(b) op-function for `op` $= \div$

`ZPlus`$(n, \mathcal{V}^n, m, \mathcal{V}^m)$
(0) *node* $res := \epsilon$;
(1) IF $(n =$ *0-node*$)$
(2) THEN $res := m$;
(3) ELSE IF $(m =$ *0-node*$)$
(4) THEN $res := n$;
(5) ELSE IF $(n, m \in \mathcal{K}_T)$ THEN
(6) $v :=$ `value`$(n) +$ `value`(m);
(7) $res :=$ `getTerminalNode`(v);
(8) RETURN res;

(c) op-function for `op` $= +$

`ZMinus`$(n, \mathcal{V}^n, m, \mathcal{V}^m)$
(0) *node* $res := \epsilon$;
(1) IF $(n = m \;\&\&\; \mathcal{V}^n = \mathcal{V}^m)$
(2) THEN $res :=$ *0-node*;
(3) IF $(m =$ *0-node*$)$
(4) THEN $res := n$;
(5) ELSE IF $(n, m \in \mathcal{K}_T)$ THEN
(6) $v :=$ `value`$(n) -$ `value`(m);
(7) $res :=$ `getTerminalNode`(v);
(8) RETURN res;

(d) op-function for `op` $= -$

in case of a terminal *1-node* a 0-sup. semantics is addressed (cf. beginning of Sec. 3.3, p. 40)–, one may only terminate the recursion, if either one of the nodes is the terminal *0-node* (line 3) or **both** are the terminal *1-node*, which is already covered in line 1, but stated here for compatibility with 0-1 *p*ZDDs once again (see discussion at the end of this section).

(2) The op-function for `op` $= \vee$ is given as Algo. 3.3.b, where similar to the above case a termination in case of reaching a terminal *1-node* is only possible, if this is the case for both *p*ZDDs and in case all variables are processed. In case of node-equality the recursion can only be terminated, if the sets of function-variables also matches (line 1-2).

(3) Due to the different interpretation of skipped variables, depending on the terminal node reached, the complement-building of *p*ZDDs is more complex than in case of BDDs. In order to avoid the negation of *p*ZDDs the op-function `ZSetMinus` is defined, so that a call to *p*`ZApply` with op-function `ZSetMinus` gives one a *z*-BDD based representation of the function $f := f_n \wedge \neg f_m$. –This is of great use when it comes to the symbolic computation of a high-level models reachability set of states (cf. Algo. 4.3, p. 90).– The pseudo-code for operator `ZSetMinus` is given as Algo. 3.3.c.

In order to increase the readability of the symbolic algorithms presented later, op-functions for boolean operators are implemented in such a way, that they are also applicable to non-0-1 *p*ZDDs. Otherwise one would need to convert a *p*ZDD to its boolean counterpart, by replacing each terminal node t, where `value`$(t) \neq 0$, with the terminal *1-node*. I.e. when executing the *p*`ZApply`-algorithm for two *p*ZDDs Z_g and Z_h and `op` $\in \{\vee, \wedge, \setminus\}$, one always obtains a 0-1 *p*ZDD representing $\mathsf{Z}_g \,\mathtt{op}\, \mathsf{Z}_h$. This is achieved by simply removing all cases from the op-functions of Algo. 3.3, where a non-terminal node is returned as result (line 1-2 of Algo. 3.3.a and line 1-6 of Algo. 3.3.b).

`op`-*functions implementing arithmetic operators*
The `op`-functions for `op` $\in \{\cdot, \div, +, -\}$ can be implemented analogously to the boolean ones, where one simply needs to return `value`(n) `op` `value`(m) in case terminal non-*0-node*s are reached (see Algo. 3.4).

3.4.3 Variants of the *p*`ZApply`-algorithm

Besides partially shared ZDDs, where the sets of the function variables of the employed *p*ZDDs may differ, one may also identify some specific scenarios, each yielding its own improved variant of the *p*`ZApply`-algorithm:

(1) Fully shared ZDDs, here the *p*ZDDs have identical sets of function variables: $\mathcal{V}^{\mathsf{A}} = \mathcal{V}^{\mathsf{B}} \subseteq \mathcal{V}^{G}$. Due to the Shannon expansion as carried out above, one may reveal that *p*ZDDs of this kind can be manipulated by a standard `ZApply` algorithm, given that a *0-maintaining* operator `op` is applied (cf. Appendix A, p. 161). Contrary to the *p*`ZApply` algorithm, the respective variant to be developed is solely required to recurse on variables, for which actually a node is allocated. This simplifies the algorithm, so that it is only required to be called with the *0-maintaining* operator to be applied together with the root nodes of the fully shared ZDDs to be manipulated. In case of a skipping of a function variable the `ZApply` algorithm follows then a *0-sup.*-rule, which means that within the `else`-branch and the skipped variable the current node is the next node to be traversed, whereas within the `then`-branch and the skipped variable the *0-node* is passed to the next call to `ZApply`.

(2) Non-shared ZDDs, here the *p*ZDDs have no function variable in common: $\mathcal{V}^{\mathsf{A}} \cap \mathcal{V}^{\mathsf{B}} = \emptyset$. They can be manipulated by a specialized *p*`ZApply`-algorithm. This variant is obtained by simply omitting line 7-9, 12-14, and 18-20 of the original *p*`ZApply`-algorithm as developed above.

The *p*`ZApply` in case of non- and partially shared ZDDs has to consider each variable the initial *p*ZDDs are defined on ($\mathcal{V}^{cube} := \mathcal{V}^{\mathsf{A}} \cup \mathcal{V}^{\mathsf{B}}$). Thus one needs to execute two recursive calls for each variable $\mathsf{v}_c \in \mathcal{V}^{cube}$. This significantly increases the number of entries for the computed table. However, as it will be discussed next, in case `op` $\in \{\wedge, \cdot\}$, the *p*`ZApply`-algorithm can be replaced a more efficient variant, which we denote as *p*`ZAnd`-algorithm.

The p`ZAnd`-*algorithm*
While skipping a variable, one assumes a *dnc* - or a *0-sup.* semantics, whether the associated variable is a function variable or not for the respective *p*ZDD-graphs. In case of non- and partially shared ZDDs this led to many case distinctions. If one assumes now that a variable v_c is skipped in both *p*ZDDs, the following scenarios appear:

(1) The omission results from different semantics, this means for the represented functions by applying the Shannon-expansion:

$$\begin{aligned} g &:= \neg\mathsf{v}_c g_0 + \mathsf{v}_c g_1 && \textit{dnc} \text{ sem., i.e.} g = g_1 = g_0 \\ h &:= \neg\mathsf{v}_c h_0 && \textit{0-sup.} \text{ sem., i.e.} h_1 \overset{(!)}{=} 0 \end{aligned}$$

This can be employed for computing $f := g \cdot h$, yielding:

$$\begin{aligned} f &:= (\neg\mathsf{v}_c g_0 + \mathsf{v}_c g_1)(\mathsf{v}_c h_1 + \neg\mathsf{v}_c h_0) \text{ with } h_1 = 0 \\ &= \neg\mathsf{v}_c g_0 h_0 + \mathsf{v}_c \neg\mathsf{v}_c g_1 h_0 \\ &= \neg\mathsf{v}_c g_0 h_0 + 0 \\ &= \neg\mathsf{v}_c g_0 h_0 \end{aligned}$$

Thus function f solely depends on the expansion of $\neg\mathsf{v}_c g_0 h_0$, where the *p*ZDD based representation of g_0 and h_0 are rooted in the 0-children of the "fictious nodes" being skipped, which are the current nodes themselves. Since furthermore v_c also fulfills the *0-sup.*-semantics nothing needs to be done here.

Algorithm 3.5 The p`ZAnd`-algorithm implementing conjunction and multiplication

p`ZAnd`(op, n, m)
(0) *node* res, e, t;
(1) $res := op(n, \emptyset, m, \emptyset)$;
(2) IF $res = \epsilon$ THEN RETURN res;

/∗ *Check* `op`*-cache if result is already known* ∗/
(3) $res =$ `cacheLookup`(op, n, m);
(4) IF $res \neq \epsilon$ THEN RETURN res;

/∗ *Depending on the order the respective recursion is entered* ∗/
(4) IF `var`$(n) =_\pi$ `var`(m) THEN
(5) $v :=$ `var`(n);
(6) $e :=$ p`ZAnd`$(op,$ `else`$(n),$ `else`$(m))$;
(7) $t :=$ p`ZAnd`$(op,$ `then`$(n),$ `then`$(m))$;
(8) ELSE IF `var`$(n) <_\pi$ `var`(m) THEN
(9) $v :=$ `var`(n);
(10) $e :=$ p`ZAnd`$(op,$ `else`$(n), m)$;
(11) IF $(n \in \mathcal{V}_\mathsf{B})$ THEN $t :=$ p`ZAnd`$(op,$ `then`$(n),$ *0-node*$)$;
(12) ELSE $t :=$ p`ZAnd`$(op,$ `then`$(n), m)$;
(13) ELSE
(14) $v :=$ `var`(m);
(15) $e :=$ p`ZAnd`$(op, n,$ `else`$(m))$;
(16) IF $m \in \mathcal{V}_\mathsf{A}$ THEN $t :=$ p`ZAnd`$(op,$ *0-node*$,$ `then`$(m))$;
(17) ELSE $t :=$ p`ZAnd`$(op, n,$ `then`(m);

/∗ *allocate new node, respecting isomorphism and 0-sup. rule* ∗/
(18) $res :=$ `getUniqueZDDNode`(v, t, e);

/∗ *Insert result into* `op`*-cache and terminate recursion* ∗/
(19) `cacheInsert`(op, n, m, res);
(20) RETURN res;

(2) The omission results from the same semantics: Nothing has to be done, since
  (2.a) under the *dnc*-semantics $\mathsf{v}_c \notin \mathcal{V}^f$ and therefore nothing needs to be done (cf. Eq. 3.11).
  (2.b) when the *0-sup.*-semantics is applicable, also nothing needs to be done, since according to Eq. 3.8 one has:

$$f = \mathsf{v}_c(0 \cdot 0) + \neg\mathsf{v}_c(h_0 \cdot g_0) = \neg\mathsf{v}_c(h_0 \cdot g_0),$$

  which is the Shannon expansion of a node to be *0-sup.* (Eq. 3.5), since the 1-cofactor of f is the constant 0-function. Consequently one solely needs to traverse the 0-children of the two "fictiously *0-sup.*-nodes" being skipped, which are the current nodes n and m themselves.

For `op` $= \{\wedge, \cdot\}$ the above conclusions allow one to omit lines 22-29 of the p`ZApply`-algorithm (cf. Algo. 3.2). I.e. in contrast to the p`ZApply`-algorithms, the p`ZAnd`-algorithm, like the algorithm for fully shared ZDDs (in case of *0-maintaining* operators), only needs to stop at levels, where nodes are actually allocated, rather than executing two recursive calls for each variable $\mathsf{v}_c \in \mathcal{V}^{cube}$. This allows one furthermore to omit the set of function variables and the cube-set *cube* as it was required in case of the p`ZApply`-algorithm. The algorithm which realizes the respective functionality is given in Algo. 3.5. As `op`-functions one may employ the `op`-functions for $\wedge$ and $\cdot$ specified above, where the set of function-variables can be replaced with the empty sets.

For exemplification, one may refer to Fig. 3.10, where on the right the call-tree for p`ZAnd` is shown, if the pZDDs A and B of Fig. 3.9 are employed. Besides the different function calls, the call tree also indicates the newly allocated nodes as created at the termination of the respective recursion. In case of the second call to p`ZAnd`$(\cdot,$ *1-node*$, n_1)$, the framed of

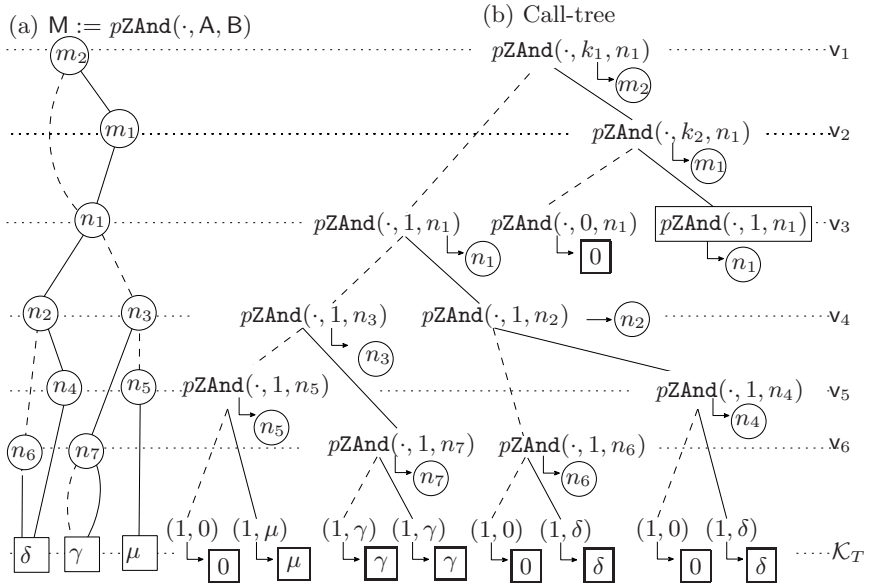


Figure 3.10: Resulting *p*ZDD and call-tree for $\mathsf{M} := \mathsf{A} \cdot \mathsf{B}$

Fig. 3.10.b, one is furthermore enabled to terminate the recursion directly, since the result is already known due to a preceding call of $p\texttt{ZAnd}(\cdot, \textit{1-node}, n_1)$. The skipping of variables under different semantics is nicely illustrated by call $p\texttt{ZAnd}(\cdot, \textit{1-node}, n_2)$, where the subsequent recursion of the `else`-branch does not stop at the level of variable v_5.

3.4.4 Relabeling of variables

Sometime it is also necessary to re-label nodes with another variable. I.e. one assigns a new variable to all nodes labeled with a specific variable label, e.g. IF $\mathsf{v} = \texttt{var}(n_k)$ THEN $\texttt{var}(n_k) := \mathsf{u}$. This functionality is addressed in the following by the notation: $\mathsf{B}\{\mathsf{v} \leftarrow \mathsf{u}\}$. The respective algorithm can be implemented straight-forward by basically traversing the *p*ZDD from bottom to top and calling $\texttt{getNode}(\mathsf{u}, \texttt{then}(n), \texttt{else}(n))$ for each node n where $\mathsf{v} = \texttt{var}(n)$. The iterative application allows one then the replacement of a set of variable labels rather than individual labels only. Since one operates on ordered sets of variables the notation $\mathsf{B}\{\vec{\mathsf{s}} \leftarrow \vec{\mathsf{t}}\}$, where $\vec{\mathsf{s}}$ and $\vec{\mathsf{t}}$ are vectors of boolean variables and have the same dimension, will be employed in the following. It is clear that the newly introduced variable labels are not allowed to be function variables of the respective *p*ZDD to be re-labeled.

3.4.5 The *p*ZRestrict-operator

Sometimes it is necessary to extract the co-factor of a function f concerning a function variable v_i and with respect to its value. This can be achieved with the *0-sup.*-variant of Bryant's *p*ZRestrict-algorithm [Bry86]. As a result, this algorithm delivers a symbolic representation of the function $f|_{\mathsf{v}_i:=b}$ for $b \in \{0, 1\}$. The basic algorithm can be extended to the case of a set of variables, rather than a single variable. As result one obtains a symbolic representation of function $f|_{\vec{v}:=\vec{b}}$ where $b_i \in \{0, 1\}$. The pseudo-code of the algorithm incorporating this behavior is specified as Algo. 3.6. As input arguments the algorithm takes the ordered tuple of variables the *p*ZDD will be restricted to $(\vec{\mathsf{v}})$, the bit-assignment to determine their values, the index of the currently processed variable (i), the root node of the *p*ZDD to be processed and the *z*-BDD *cube*, where the latter encodes the cube set over $\mathcal{V}^f$.

3.4.6 The *p*ZAbstract-operator

The abstraction from a set of function variables $\vec{\mathsf{v}}$ is implemented by the *p*ZAbstract-algorithm, called with a respective `op`-function. While traversing the original *p*ZDD and

Algorithm 3.6 The pZRestrict-algorithm

pZRestrict$(\vec{\mathtt{v}}, \vec{b}, i, n, cube)$
(0) *node* t, e, res;

/* *Reached terminal nodes, end of recursion* */
(1) IF $cube \in \mathcal{K}_T \parallel i = \dim(\vec{b}) + 1$ THEN RETURN n;

/* *Check* op*-cache if result is already known* */
(2) $res :=$ cacheLookup$(p$ZRestrict$, \vec{\mathtt{v}}, \vec{b}, i, n, cube)$;
(3) IF $res \neq \epsilon$ THEN RETURN res;
(4) $\mathtt{v}_i := \vec{\mathtt{v}}[i]$;

/* *variable to be removed not reached yet* */
(5) IF $\mathtt{v}_i \ {}_\pi\!> \mathtt{var}(n)$ THEN
(6) WHILE $\mathtt{var}(n) \ {}_\pi\!\geq \mathtt{var}(cube)$ DO $cube :=$ then$(cube)$; END
(7) $t := p$ZRestrict$(\vec{\mathtt{v}}, \vec{b}, i,$ then$(n), cube)$;
(8) $e := p$ZRestrict$(\vec{\mathtt{v}}, \vec{b}, i,$ else$(n), cube)$;
(9) $res :=$ getUniqueZDDNode$(\mathtt{var}(n), t, e)$;

/* *Reached variable to be removed* */
(10) ELSE IF $\mathtt{v}_i =_\pi \mathtt{var}(cube)$ THEN

/* *Variable to be removed is* 0-sup. */
(11) IF $\mathtt{var}(n) \ {}_\pi\!> \mathtt{var}(cube)$ THEN
(12) IF $b_i = 0$ THEN $res := p$ZRestrict$(\vec{\mathtt{v}}, \vec{b}, i{+}1, n, cube)$

/* *Found a node labeled with variable to be removed* */
(13) ELSE $res :=$ *0-node*
(14) ELSE
(15) $cube :=$ then$(cube)$;
(16) IF $b_i = 0$ THEN $res := p$ZRestrict$(\vec{\mathtt{v}}, \vec{b}, i{+}1,$ else$(n), cube)$
(17) ELSE $res := p$ZRestrict$(\vec{\mathtt{v}}, \vec{b}, i{+}1,$ then$(n), cube)$

/* *Insert result into* ZRestrict*-cache and terminate recursion* */
(18) cacheInsert$(p$ZRestrict$, \vec{\mathtt{v}}, \vec{b}, i, n, cube, res)$;
(19) RETURN res;

eliminating nodes labeled with $\mathtt{v}_i \in \vec{\mathtt{v}}$, it might appear that previously distinct subtrees collapses. In such cases the respective version of the pZApply-algorithm with the binary boolean operator op has to be called. Consequently the pZAbstract-algorithm constructs a representation of function $f|_{\mathtt{v}_i}\,\mathtt{op}\,f|_{\neg\mathtt{v}_i}$, where in case of $\mathtt{op} \in \{\vee, +\}$ the existential and in case of $\mathtt{op} \in \{\wedge, \cdot\}$ the universal quantification is realized. The pZAbstract-algorithm can be extended to the case of handling sets of variables.

The pseudo-code of the pZAbstract-algorithm is given as Algo. 3.7. Its parameters are the binary operator op for steering the merging of collapsing subtrees, the set of variables to be abstracted from (the ordered tuple $\vec{\mathtt{v}}$), the position of the current variable to be removed within $\vec{\mathtt{v}}$ (i), the root node of the pZDD to be manipulated (n), and the z-BDD cube representing the set of function variables of the pZDD to be manipulated. In line 1 one tests if the terminal condition for terminating the recursion is satisfied. If this is the case a respective node is returned, otherwise one tests at first, if a result from a previous recursion is known (line 2-3). In case the cache-lookup does not deliver such a result the recursion is entered, where three different cases must be covered (line 7-18): (a) The pseudo-code of line 8-10 covers the case that a *0-sup.* variable is to be removed. (b) The pseudo-code of line 11-13 covers the case that the current node is labeled with the variable to be removed. In both cases the removal of variables within the current paths yields a collapsing of subtrees, even in case of *0-sup.* variables! Thus it is now required to execute the pZApply-algorithm for merging the sub-trees. However, since the collapsing subtrees rooted in node t and e are fully shared ones, i.e. they are both defined one the same set of variables, namely the ones

Algorithm 3.7 The *p*ZAbstract-algorithm

```
pZAbstract(op, v⃗, i, n, cube)
(0)   node  t, e, res;

/* Reached terminal nodes, end of recursion */
(1)   IF (cube ∈ K_T || i = dim(b⃗) + 1) THEN RETURN res;

/* Check op-cache if result is already known */
(2)   res := cacheLookup(pZAbstract, op, v⃗, i, n, cube);
(3)   IF res ≠ ε THEN RETURN res;

(4)   v_i := v⃗[i];
(5)   n := var(n);

/* Variable to be abstracted is located below var(cube) */
(6)   WHILE v_i ≺_π var(cube) DO cube := then(cube); END

/* Reached variable to be abstracted */
(7)   IF n ≽_π v_i THEN

/* Variable to be abstracted is 0-sup. */
(8)       IF v_i ≠_π n THEN
(9)           t := 0-node;
(10)          e := pZAbstract(op, v⃗, i+1, n, then(cube));

/* Reached node carrying variable to be abstracted */
(11)      ELSE
(12)          t := pZAbstract(op, v⃗, i+1, then(n), then(cube));
(13)          e := pZAbstract(op, v⃗, i+1, else(n), then(cube));

/* Merge collapsing paths */
(14)      res := pZApply(...);


/* Reached node carrying variable not to be abstracted */
(15)  ELSE
(16)      t := pZAbstract(op, v⃗, i, then(n), cube);
(17)      e := pZAbstract(op, v⃗, i, else(n), cube);
(18)      res := getUniqueZDDNode(n, t, e);

/* Insert result into pZAbstract-cache and terminate recursion */
(19)  cacheInsert(pZAbstract, op, v⃗, i, n, cube, res);
(20)  RETURN res;
```

represented by *cube*, one is enabled to employ the more efficient ZApply-algorithm, in case a *0-maintaining* operator is applied. Otherwise the standard *p*ZApply-algorithm, with the respective set of parameters (omitted here for simplicity) must be called (line 14). (c) The pseudo-code of line 15-18 covers the case, that the algorithm reached a node referring to a variable not to be abstracted.

In case op $\in \{\vee, +\}$ the algorithm can be simplified, since here one only needs to take care of the nodes, which are labeled with variables to be abstracted. The handling of *0-sup.* variables to be abstracted is not necessary. Contrary to this, in case op $\in \{\wedge, \cdot\}$ this is not possible, since each path where $\mathtt{v}_i$ is *0-sup.* must be evaluated to 0.

3.5 Applications

In the following, we will introduce the basics for efficiently employing ZDDs within symbolic state graph representation. Such state graph representations are commonly derived from some high-level stochastic model description, which is irrelevant for the moment. The obtained stochastic transition relation as represented by a *p*ZDD directly encode the transition rate matrix of a CTMC. Consequently numerical solvers may employ the *p*ZDD based matrix representation for computing state probabilities of the high-level stochastic model.

Algorithm 3.8 Generating symbolic representations of singletons

```
Encode(b_1, ..., b_{B_S}, c, v⃗)
(0)  l := B_S;
(1)  n := getTerminalNode(c);

/* Construct ZDD bottom up */
(2)  WHILE l > 0 DO
(3)      IF b_l = 1 THEN n := getUniqueZDDNode(v⃗[i], n, 0-node);
(4)      l--;
(5)  END

(6)  RETURN n;
```

In this section concepts as known from ADD based performability analysis, i.e. mainly the work presented in [Par02] will be extended to the case of *p*ZDDs. Details of the implementation of the new ZDD based solvers for computing state probabilities are not presented here, they can be found in [Zim05, Har06]. Since the exact BDD type is often without significance, we will sometimes generically speak of BDDs, *Mt*-DDs, or decision diagrams (DDs). In case of the former either *dnc-free* BDDs or *z*-BDDs are addressed. In case of *Mt*-DDs their multi-terminal extensions, such as ADDs, or *p*ZDDs is referred to. In case all types of BDDs and *Mt*-DDs is referred to, we will speak generically of DDs. Since we do not focus on the application of binary operators onto the DDs, the set of their function variables is without concern in the following, consequently instead of *p*ZDDs we will generically speak of ZDDs. But nevertheless, one may keep in mind that their sets of function variables may differ or be actually disjoint as discussed in case of cross-product building of *Mt*-DDs.

3.5.1 ZDD based representations of sets and relations

We briefly introduce the encoding of states and transition relations as employed under state space based analysis of high-level model description methods [Bry92, HMKS99].

In the scope of this work, finite sets of states $\mathbb{S}$ are considered, where each state is a vector of integers. Each position within these state vectors refers to one of the N SVs, so that $\vec{s}[i]$ gives one value of SV $\mathfrak{s}_i$ with respect to state $\vec{s}$. The number of bits required for encoding each state is defined as follows:

$$B_{\mathbb{S}} := \sum_{i:=1}^{N} B_i, \text{ where } B_i := \lceil \log_2(K_i) \rceil, \text{ with } K_i := \max_{\vec{s} \in \mathbb{S}}(\vec{s}[i]). \tag{3.12}$$

Based on this, the elements of $\mathbb{S}$ can be encoded as binary assignments of fixed length, if a respective function $\mathcal{E} : \mathbb{S} \mapsto \mathbb{B}^{B_{\mathbb{S}}}$ is provided. The DD accepting only the boolean assignments as obtained from $\mathbb{S}$ as its satisfaction set, gives a graph based representation of the characteristic function of $\mathbb{S}$. It is clear that the desired DD can be generated iteratively, so that a single element of $\mathbb{S}$ is encoded as ZDD by employing Algo. 3.8. Since this algorithm is employed in the next chapter for encoding states and transitions, it is useful to explain some of its implementation details now: As input parameters algorithm `Encode` takes a bit-string of length $B_{\mathbb{S}}$, the function value to be stored within the terminal node, and the ordered tuple of length $B_{\mathbb{S}}$, containing the function variables of the DD. Its main idea is that for 0-assigned bit positions nothing needs to be done, where in case of 1-assigned bit positions a node must be allocated at the respective level (line 3). Each call to `Encode` gives one then a ZDD based representation of a single element of $\mathbb{S}$. The union over all singletons $\mathbb{S}$ is made of, yields the desired symbolic representation of $\mathbb{S}$, so that the following definition holds:

Definition 3.11: Symbolic Representation of Sets

The DD S over the variables $\vec{\mathsf{v}}$ and rooted in a node n is a symbolic representation of a set $\mathbb{S}$, i.e.

$$\mathsf{S} \equiv \mathbb{S} \Leftrightarrow \texttt{Satisfy}(\mathcal{E}(\vec{s}), n, \vec{\mathsf{v}}) = \begin{cases} 1 \text{ iff } \vec{s} \in \mathbb{S} \\ 0 \text{ else} \end{cases}$$

where `Satisfy` is the function as defined in Sec. 3.2.

At this state it is necessary to mention the main advantages of employing ZDDs, when it comes to the encoding of sets:

(1) 0-assigned bit positions within $\vec{b}$: here function $\texttt{Encode}(\vec{b}, c, \vec{\mathsf{v}})$ needs to do nothing.

(2) The values K_i are in general not known prior to set generation. Thus during SG generation, one is forced to allocate a new most significant bit for any SV $\mathfrak{s}_i$ by simply declaring a new Boolean variable for the respective DD Z. Contrary to a *dnc*-free DD, a *0-sup.* DD has the nice feature that the structure of its graph does not need to be changed, since the newly allocated variable is 0-assigned for all previously encoded states. Thus it is not necessary to know the maximum K_i of SV $\mathfrak{s}_i$ in advance. However, for *dnc*-free DDs one is forced to insert a 0-assigned node on each path for the newly allocated variable. This can be achieved by executing $\mathsf{S} := \mathsf{S} \cdot \mathsf{B}_{\mathsf{v}_i=0}$, with $\mathsf{B}_{\mathsf{v}_i=0} := \texttt{Encode}(0, 1, \mathsf{v}_i)$, where $\mathsf{B}_{\mathsf{v}_i=0}$ represents function $f(\mathsf{v}_i) = 1 \Leftrightarrow \mathsf{v}_i = 0$.

A *sLTS* defines a set of transitions, so that $(a, s, t, \mu) \in T$ holds, if there is a directed edge labeled with activity label a and rate μ and connecting state s with state t. In the scope of this work the components s and t of the elements of T are elements of $\mathbb{S}$. The activity label a is an element of a finite set of labels $\mathcal{A}ct$ and the stochastic rate $\mu \in \mathbb{R}^+$. If one defines now an adequate boolean encoding function $\mathcal{E} : \mathcal{A}ct \times \mathbb{S} \times \mathbb{S} \mapsto \mathbb{B}^{n_\mathcal{V}}$ for the elements of T, the symbolic representation of a transition relation by a DD rooted in a node n, so that $T \equiv \mathsf{T}$, is analogous to the above explained method of symbolic set representation. The rate μ of each transition is stored within a terminal node, so that $\texttt{Satisfy}(\mathcal{E}(a, \vec{s}, \vec{t}), n, \vec{\mathsf{v}}) = \mu$ for each stochastic transition $(a, s, t, \mu) \in T$. In terms of function `Encode`, one only needs to pass the rate as second argument (parameter c in Algo. 3.8). In case one intends to symbolically represent a probabilistic LTS (*pLTS*) T^i the process can be handled analogously. However, in such a setting the values of the terminal nodes are interpreted as probabilities, rather than rates, i.e. $\mu \in (0, 1]$. In case of an extended *sLTS* (*esLTS*) T, the situation is more complicated, since one needs to distinguish between stochastic rates and probabilities. A case distinction can easily be achieved by assigning a leading minus sign to the probabilities or rates, and storing the other values as before. Alternatively one could also store the *pLTS* and *sLTS* as contained in the *esLTS* separately, one may refer to [Sie01] for details.

So far it was quietly assumed, that the set $\mathcal{V}$ consists of the variables $\mathsf{v}_1, \ldots \mathsf{v}_{n_\mathcal{V}}$, however for simplicity different sets of boolean variables will be introduced now. For binary encoding $(a, \vec{s}, \vec{t}, \mu) \in T$ under a *"most-significant-bit-first"* order, the following ordered sets of boolean variables are defined:

Definition 3.12: Boolean variables holding binary encoded transitions

(1) $\vec{\mathsf{a}} := (\mathsf{a}_1, \ldots, \mathsf{a}_{B_{\mathcal{A}ct}})$ holding the values for the binary encodings of the activity labels,

(2) $\vec{\mathsf{s}}^i := (\mathsf{s}^i_1, \ldots, \mathsf{s}^i_{B_i})$ holding the values of the binary encodings of $\vec{s}[i]$, and

(3) $\vec{\mathsf{t}}^i := (\mathsf{t}^i_1, \ldots, \mathsf{t}^i_{B_i})$ holding the values of the binary encodings of $\vec{t}[i]$

The s- and t-variables are collected as two ordered tuples, where a most-significant bit first order is assumed, yielding:

$$\begin{array}{l}\vec{\mathsf{s}} := (\mathsf{s}_1, \ldots, \mathsf{s}_m) := (\mathsf{s}^1_1, \ldots, \mathsf{s}^1_{B_1}, \ldots, \mathsf{s}^n_1, \ldots \mathsf{s}^n_{B_n}) \text{ and} \\ \vec{\mathsf{t}} := (\mathsf{t}_1, \ldots, \mathsf{t}_m) := (\mathsf{t}^1_1, \ldots, \mathsf{t}^1_{B_1}, \ldots, \mathsf{t}^n_1, \ldots \mathsf{t}^n_{B_n}).\end{array} \tag{3.13}$$

For convenience also the symbol $\mathcal{V}_{\mathcal{A}ct}$, $\mathcal{V}_{\mathsf{s}}$ and $\mathcal{V}_{\mathsf{t}}$ will be employed, if referring directly to the sets of boolean a-, s- and t-variables.

The size of a DD is known to be sensitive concerning the chosen ordering on the boolean variables the DD is defined on. For keeping the DDs small, we define that the boolean vector $\vec{\mathsf{a}}$, the variables of which encode the activity labels, appears first. Starting at level $B_{\mathcal{A}ct} + 1$ the boolean vectors holding the binary encoded source and target states of a transition follow in an interleaved fashion, defined as follows:

$$\mathsf{s}_1 <_\pi \mathsf{t}_1 <_\pi \mathsf{s}_2 <_\pi \ldots <_\pi \mathsf{s}_m <_\pi \mathsf{t}_m \tag{3.14}$$

This ordering scheme is a commonly accepted heuristics for obtaining small DD sizes in the context of symbolic representations of transition systems [EFT93, FM97, Sie01]. As it will be discussed below, the interleaved ordering seems to be especially of great value, when it comes to the encoding of identity matrices, where an advantage for *0-sup*. DD-types over the *dnc* based counterparts will be pointed out.

3.5.2 ZDD based representations of matrices

Since ZDDs map boolean vectors of the same dimension to an arbitrary set of values, they are highly suited for representing real-valued matrices. One simply needs to binary encode the set of row and column indices and store the matrix entry within the terminal nodes.

Basic encoding scheme

A matrix M can be understood as a two-dimensional finite function $M : R \times C \mapsto \mathbb{D} \subset \mathbb{R}$, where $R, C \subset \mathbb{N}_0^+$ are the sets of row and column indices. The main idea of representing a real-valued matrix M is as follows: One simply binary labels the set of row indices R as well as the set of column indices C, by once again employing an adequate binary mapping $\mathcal{E}$:

$$\mathcal{E} : R \times C \mapsto \mathbb{B}^{B_r + B_c},$$

where $B_r \geq \lceil \log_2(|R|) \rceil$ is the number of bits required for encoding the row indices and where $B_c \geq \lceil \log_2(|C|) \rceil$ is the number of bits required for encoding the column indices. Each bit position as produced by $\mathcal{E}$ is bijectively mapped to a variable of $\mathcal{V}$. Since for each pair of row and column indices a real-valued entry $\in \mathbb{D} := \{M(r,c) | r \in R \wedge c \in C\}$ exists, and since the above (ordered) encoding scheme $\mathcal{E}$ together with the different matrix entries ($\mathbb{D}$) define a pseudo boolean function: $\mathbb{B}^{B_c + B_r} \mapsto \mathbb{D}$, the encoding of a matrix by a ZDD is achieved:

Definition 3.13: ZDD based representation of a real-valued (quadratic) matrix

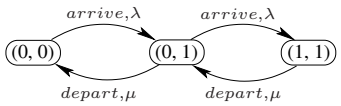
---

A ZDD Z over $< \mathcal{V}, \pi, \mathbb{D} >$ rooted in a node k is a canonical representation of a real-valued $(2^n \times 2^n)$ matrix M concerning a fixed order π ($M \equiv_\pi \mathsf{M}$), if the following three conditions hold:

$$(i)\, M(r,c) \leftrightarrow \mathbb{D},\ (ii)\, R \times C \leftrightarrow \mathbb{B}^{2n} \text{and} (iii)\, \mathtt{Satisfy}(\mathcal{E}(r,c), k, \vec{\mathsf{v}}) = M(r,c), \text{where} 2n := |\mathcal{V}|.$$

According to the above definition, the represented matrices are quadratic, and the size of the set of row and column indices (R, C) is a power of 2. If this is not the case, one simply needs to extend M by an adequate number of rows and columns filled with zeroes (0-padding [EFT93]), where we will refer to such entries as *dummy entries*.
For interpreting the ZDD based representation of stochastic transition system directly as its transition rate matrix, it is necessary to distinguish, whether a bit position refers to a binary encoded row or column index. Therefore we define that the variable s_i refers to bit position

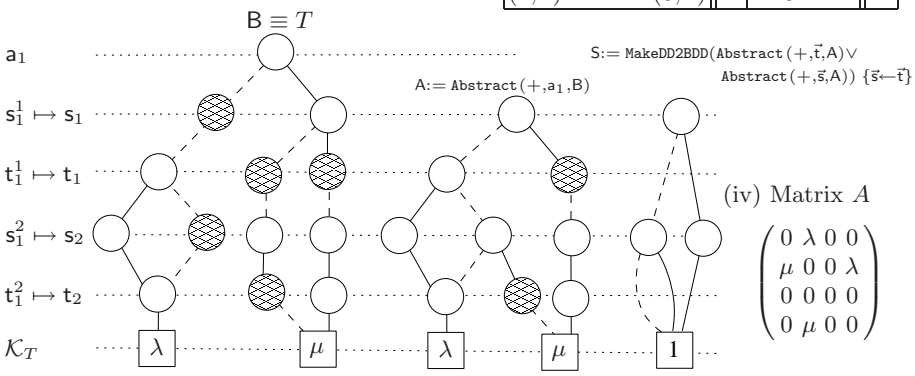
(i) STLS T



(ii) Binary encodings

$\vec{s} \xrightarrow{l,q} \vec{t} \in T$	a_1	s_1^1 t_1^1 s_1^2 t_1^2	f_B
$(0,0) \xrightarrow{arrive,\lambda} (0,1)$	0	0 0 0 1	λ
$(0,1) \xrightarrow{arrive,\lambda} (1,1)$		0 1 1 1	
$(0,1) \xrightarrow{depart,\mu} (0,0)$	1	0 0 1 0	μ
$(1,1) \xrightarrow{depart,\mu} (0,1)$		1 0 1 1	

(iii) pZDD-based representations



(iv) Matrix A

$$\begin{pmatrix} 0 & \lambda & 0 & 0 \\ \mu & 0 & 0 & \lambda \\ 0 & 0 & 0 & 0 \\ 0 & \mu & 0 & 0 \end{pmatrix}$$

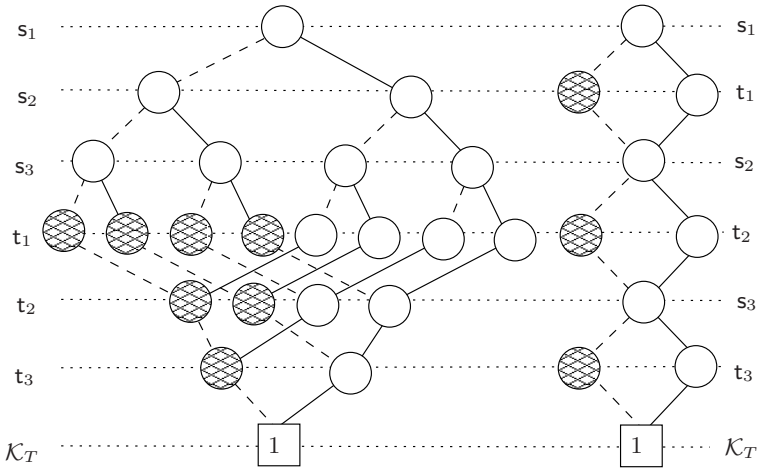
Figure 3.11: *sLTS*, its ZDD based representation and underlying transition rate matrix

i of a binary encoded row index r and that variable t_j refers to bit position j of a binary encoded column index c, where a *most-significant-bit-first* and *left-to-right* sequence of bits is assumed (cf. Eq. 3.14). *The connection to the ZDD based encoding of a stochastic transitions system is clear, one abstracts from the boolean variables encoding the activity labels, i.e. from the* a*-variables and simply interprets the boolean variables collected in the vector* $\vec{\mathsf{s}}$ *and* $\vec{\mathsf{t}}$ *(cf. Eq. 3.13) as binary encoded row - and column indices.* As a consequence of this encoding scheme each non-reachable state induces a *dummy entry* within the symbolic represented transition rate matrix. However, the generated transition rate matrices are always square matrices, since $B_r = B_c = B_\mathbb{S}$, with $|R| = |C| = |\mathbb{S}|$ holds. Furthermore their dimension is automatically a power of 2, since the number of bits (n) required for encoding a row-, column index resp. is determined by the number of bits employed for encoding the state labels, i.e. $n = n_\mathcal{V}/2$, where $n_\mathcal{V} = 2B_\mathbb{S}$ holds. As a consequence additional zero-padding as described above is unnecessary.

For exemplification one may refer now to Fig. 3.11. Part (i) depicts a small *sLTS* T, where each state is given by a 2-dim. vector. The binary encodings of this transition system is given in table 3.11.ii. This function table of a pseudo-boolean function enables one to construct the respective *Mt*-DD B, where the striped nodes are the *0-sup.* ones (Fig. 3.11.iii). If one now abstracts form the activity labels, one obtains the *Mt*-DD A. The *Mt*-DD A directly encodes the underlying transition rate matrix of the initial *sLTS* T, where the matrix is given in Fig. 3.11.iv. For deriving the matrix correctly, the SVs are mapped to the respective variables encoding the column and row-indices as it was already defined in Eq. 3.13 (p. 59). This mapping is depicted on the left hand side of Fig. 3.11.iii.
From the *sTS* as encoded by A one may derive also the BDD S representing the set of reachable states. The construction of S is hereby achieved via DD-manipulations, so that the binary encoded state labels are solely stored within the s-variables, where the t-variables are not function variables of S. As one can deduce from S, the state 10 is non-reachable. As a consequence, the third column and row of matrix A are therefore filled with zeros only.

Efficiently encoding identity matrices

The symbolic representation of the identity function is defined as follows:



(A) Non-interleaved order (B) Interleaved order

Figure 3.12: *Mt*-DD based representation of the identity function $\mathbb{1}(\mathcal{V})$

Definition 3.14: BDD based representation of the identity function

Let $\mathcal{U} := \{\mathtt{s}_1, \ldots, \mathtt{s}_n, \mathtt{t}_1, \ldots, \mathtt{t}_n\}$. The identity function on n s- and n t-variables is defined as follows: $f_{\mathbb{1}}(\mathtt{s}_1, \ldots, \mathtt{s}_n, \mathtt{t}_1, \ldots, \mathtt{t}_n) := \prod_{i:=1}^{n} \mathtt{s}_i \leftrightarrow \mathtt{t}_i$. Its symbolic representation will be denoted as $\mathbb{1}(\mathcal{U})$ in the following.

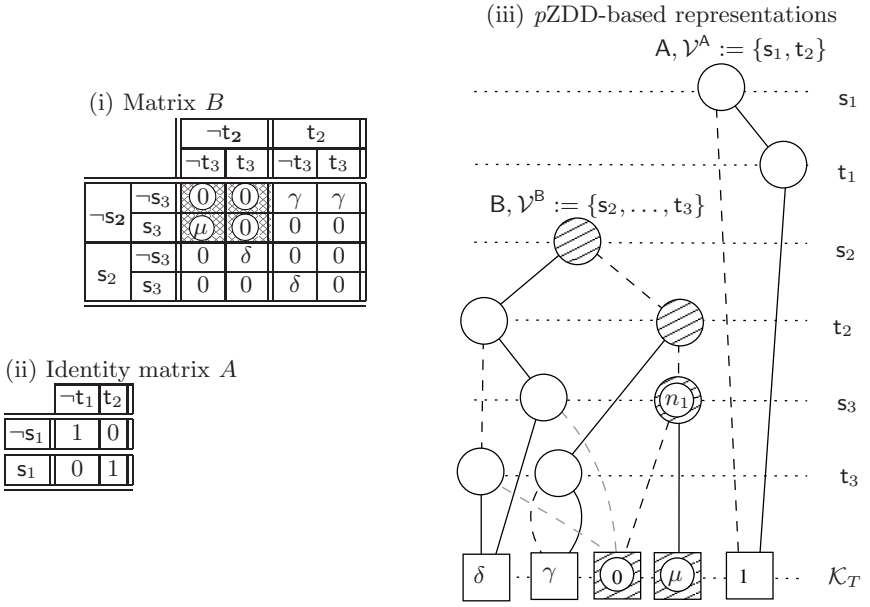
Within the context of Def. 3.13 one may interpret $\mathbb{1}(\mathcal{V})$ as an $(2^n \times 2^n)$ identity matrix. For the BDD based representation of such an identity matrix, the number of nodes is linear with respect to n, if an interleaved ordering is employed. In contrast a non-interleaved order of the variables $\mathcal{V}_\mathtt{s}$ and $\mathcal{V}_\mathtt{t}$ yields an exponential number of nodes with respect to n, where for different BDD-types the explicit number of nodes of course differs: (a) For a dnc-*free* BDD based representation of a $(2^n \times 2^n)$ identity matrix the *non-interleaved* order requires $\mathtt{sizeof}(\mathbb{1}(\mathcal{V})) = 3 \cdot 2^n - 1$ nodes. In contrast the interleaved-order is only linear in space, so that here solely $\mathtt{sizeof}(\mathbb{1}(\mathcal{V})) = 3n + 2$ nodes to be allocated (cf. [Sie01]). (b) For a *z*-BDD based representation one obtains $\mathtt{sizeof}(\mathbb{1}(\mathcal{V})) = 2(n + 1)$ for the interleaved order of the $2n$ variables. In contrast a non-interleaved order of the s– and t-variables requires then $\mathtt{sizeof}(\mathbb{1}(\mathcal{V})) = 2^{n+1}$ nodes to be allocated, which is smaller than the *dnc*-case but still exponential with respect to n. For exemplification one may refer to Fig. 3.12, which shows $\mathbb{1}(\mathcal{V})$ as ADD for the non-interleaved and interleaved ordering schemes. The hatched nodes are hereby the nodes to be eliminated under the *0-sup.* reduction rule.

Access-pattern to the matrix entries

Since we defined a *"most-significant-bit-first"* order as well as an interleaved ordering of the binary encoded row - and colum indices, a dfs traversal on the *Mt*-DDs realizes a block-wise access-pattern to the elements of the represented matrices. I.e. the boolean expansion for variable $\mathtt{s}_1$ gives: $f_M := \mathtt{s}_1 f_1^M + \neg\mathtt{s}_1 f_0^M$, where the respective (sub-)graphs of $f_{\{0,1\}}^M$, gives one the upper - or lower half of the matrix M. The subsequent expansion of $\mathtt{t}_1$ gives one then the individual quadrants of M. I.e. the boolean expansion for the first pair of variables $\mathtt{s}_1, \mathtt{t}_1$ gives:

$$f_M := \mathtt{s}_1\mathtt{t}_1 f_{11}^M + \mathtt{s}_1\neg\mathtt{t}_1 f_{10}^M + \neg\mathtt{s}_1\mathtt{t}_1 f_{01}^M + \neg\mathtt{s}_1\neg\mathtt{t}_1 f_{00}^M,$$

so that $f_{i,j}^M$ is one of the four quadrants of matrix M. In the following $f_{\vec{b}}^M$ addresses the co-factors by making the variables associated with the bit positions of $\vec{b}$ constant. Each graph rooted in node representing $f_{\vec{b}}^M$ is obviously a symbolic representation of sub-matrix $M_{i,j}$,



(i) Matrix B

		$\neg\mathsf{t}_2\,\neg\mathsf{t}_3$	$\neg\mathsf{t}_2\,\mathsf{t}_3$	$\mathsf{t}_2\,\neg\mathsf{t}_3$	$\mathsf{t}_2\,\mathsf{t}_3$
$\neg\mathsf{s}_2\,\neg\mathsf{s}_3$		0	0	γ	γ
$\neg\mathsf{s}_2\,\mathsf{s}_3$		μ	0	0	0
$\mathsf{s}_2\,\neg\mathsf{s}_3$		0	δ	0	0
$\mathsf{s}_2\,\mathsf{s}_3$		0	0	δ	0

(ii) Identity matrix A

	$\neg\mathsf{t}_1\,\mathsf{t}_2$	$\mathsf{t}_1\,\mathsf{t}_2$
$\neg\mathsf{s}_1$	1	0
s_1	0	1

Figure 3.13: Block-wise access by dfs-traversal

with $\mathcal{E}(i,j) := \vec{b}$. In the context of quadratic matrices, such a sub-matrix is also known as block-entry of the overall matrix M. The above access scheme is then applied once again for each matrix $M_{i,j}$, until one reaches the level of terminal nodes. So for a (4×4) matrix M this would give us the matrix elements $m_{r,c}$ in the following sequence: $m_{0,0}, m_{0,1}, m_{1,0}, m_{1,1}$, which are the matrix entries of the upper left quadrant of M. The next elements to follow are $m_{0,2}, m_{0,3}, m_{1,2}, m_{1,3}$, which are the elements of the upper right quadrant, and so on.

For exemplification one may refer to Fig. 3.13 which shows two matrices and their *p*ZDD based representations. In order to illustrate the block-wise-addressing scheme as realized by the chosen variable ordering, we wrote the matrices as tables, and equipped them with the boolean variables. The minterm $\neg\mathsf{s}_2\neg\mathsf{t}_2$, which is induced by the diagonally hatched path starting at the root node of the *p*ZDD B, gives us the (sub-)*p*ZDD rooted in the diagonally hatched node n_1 at level s_3. The (sub-)*p*ZDD rooted in n_1 represents the (sub-)function $f_{00}^{M}(\mathsf{s}_3, \mathsf{t}_3)$ as obtained from f_{B} by making the variables $\mathsf{s}_2, \mathsf{t}_2$ constant, here 0. Concerning the matrix this means that we extracted the upper left block-matrix $B(0,0)$ (the hatched one) as contained in the overall matrix B (cf. Fig. 3.13.i). The matrix-entries of $B(0,0)$ are then given by the values of the terminal nodes reachable from n_1, including the *0-entries* we ignored so far. A dfs-traversal with `else`-edge-first delivers the sequence 0, 0, μ, 0 of matrix entries.
The *p*ZDD A represents a (2×2) identity matrix (cf. Fig. 3.13.ii) or the boolean function $\mathsf{s}_1 \mapsto \mathsf{t}_1$. A dfs-traversal with `then`-edge-first rule delivers the sequence 1, 0, 0, 1 of matrix entries, starting at the lower right corner of matrix A.

Operating with *p*ZDD based matrix representations

The solvers for computing state probabilities considered in this work follow the hybrid technique as introduced for ADDs in [Par02]. Within this approach "pure"-symbolic matrix-matrix - and matrix-vector operations are therefore not relevant, with one exception.[1] Within a *p*ZDD based matrix representation the operation $\mathsf{M} := \mathsf{A} \cdot \mathsf{B}$ in case $\mathcal{V}^{\mathsf{A}} \cap \mathcal{V}^{\mathsf{B}} = \emptyset$ gives the cross-product of the structures A and B. This operation and its relation to the *Kronecker product* of matrices will be investigated now, as it is from great importance for the SG based composition of models (cf. Sec. 2.5 and Sec. 4.3.3).

[1] An overview over algorithms on ADD based matrix operations can be found in [Sie01, ADD97].

Given that $A \equiv_\pi \mathsf{A}$, $B \equiv_\pi \mathsf{B}$, and $\mathcal{V}^{\mathsf{A}} \cap \mathcal{V}^{\mathsf{B}} = \emptyset$, the operation $\mathsf{M} := \mathsf{A}\ \mathtt{op}\ \mathsf{B}$, for $\mathtt{op} \in \{\wedge, \cdot\}$ gives a DD M, which encodes $2^{\mathcal{V}^{\mathsf{A}}+\mathcal{V}^{\mathsf{B}}}$ possible assignments. I.e. one constructs a *Mt*-DD, which encodes the cross product of the satisfaction set of A and B. On the level of matrix-operations one might think that this realizes the *Kronecker product* $A \otimes B$. If this is truly the case will be investigated thoroughly now.

The variable order π defined on M turns out to be crucial. Thus one needs to distinguish between (a) a non-interrupted ordering scheme, i.e. the variables of $\mathcal{V}^{\mathsf{A}}$ are all smaller or larger than the ones of $\mathcal{V}^{\mathsf{B}}$ with respect to π, or (b) an interrupted ordering scheme with respect to π, i.e. the variables of $\mathcal{V}^{\mathsf{A}}$ and $\mathcal{V}^{\mathsf{B}}$ are somehow $(\mathsf{s}_i, \mathsf{t}_i)$-wise nested.

Lemma 3.6: *Let* A, B *be two symbolic representations of the matrices* A *and* B *with a fixed ordering* $\pi_{\mathsf{A},\mathsf{B}}$ *of their variables: where* $\mathsf{s}_i^{\mathsf{A}} \in \mathcal{V}^{\mathsf{A}}$ *and* $\mathsf{s}_k^{\mathsf{B}}, \mathsf{s}_l^{\mathsf{B}} \in \mathcal{V}^{\mathsf{B}}$. *Let* Mt-*DD* $\mathsf{M} := \mathsf{A} \cdot \mathsf{B}$, *depending on the* **fixed** *variable order* π *one obtains the following results:*

(1) Non-interrupted order:

(1.a) $\mathsf{s}_i^{\mathsf{A}} <_\pi \mathsf{s}_k^{\mathsf{B}}$*:* $\mathsf{A} \cdot \mathsf{B}$ *gives a symbolic representation of the* Kronecker-*product* $A \otimes B$, *i.e.:*

$$\mathsf{A} \cdot \mathsf{B} \equiv_\pi A \otimes B$$

(1.b) $\mathsf{s}_k^{\mathsf{B}} {}_\pi{>} \mathsf{s}_i^{\mathsf{A}}$ *here* $\mathsf{A} \cdot \mathsf{B}$ *gives a symbolic representation of the* Kronecker-*product* $B \otimes A$, *i.e.*

$$\mathsf{A} \cdot \mathsf{B} \equiv_\pi B \otimes A$$

(2) Interrupted order: $\mathsf{s}_i^{\mathsf{A}} <_\pi \mathsf{s}_k^{\mathsf{B}} \wedge \mathsf{s}_i^{\mathsf{A}} {}_\pi{>} \mathsf{s}_l^{\mathsf{B}}$.
Here the symbolic matrix representation of $\mathsf{A} \cdot \mathsf{B}$ *does not encode the* Kronecker-*product of the matrices* A *and* B *in general:*

$$\mathsf{A} \cdot \mathsf{B} \not\equiv_\pi A \otimes B \wedge \mathsf{A} \cdot \mathsf{B} \not\equiv_\pi B \otimes A$$

Proof: If one considers the block-wise addressing scheme as realized by *Mt*-DDs (see discussion above) and the definition of the Kronecker product, the proof of the above lemma is straight forward: If $M_{i,j}(k,l)$ addresses the matrix element in block (i,j) at position (k,l), for all elements of M we have: $M_{i,j}(k,l) = A(i,j) \cdot B(k,l)$ (cf. Sec. A.3). I.e. under the *Kronecker product* one combines the matrices A and B in an element-to-block-wise manner, where each block is equal to matrix B, but multiplied with matrix entry $a_{i,j} := A(i,j)$. The combination of the symbolically represented matrices as achieved by the `Apply`-algorithm or its variants does so for the non-interrupted case, however in the interrupted case, the row-to-row and column-to-column combination of the indices is somehow mixed.

Non-interrupted order of the variables

The order π as defined on M gives the assignment $\vec{b} := (\vec{b}_1, \vec{b}_2)$, where $\vec{b}_1$ defines the assignment of the variables of $\mathcal{V}^{\mathsf{A}}$ and $\vec{b}_2$ the assignment of the ones of $\mathcal{V}^{\mathsf{B}}$. The boolean expansion applied on the variables of $\mathcal{V}^{\mathsf{A}}$ yields:

$$\begin{aligned} f_M := \quad & \mathsf{s}_1^{\mathsf{A}}\mathsf{t}_1^{\mathsf{A}} \ldots \mathsf{s}_{n_{\mathsf{A}}}^{\mathsf{A}}\mathsf{t}_{n_{\mathsf{A}}}^{\mathsf{A}} f_{\mathsf{A}}(11\ldots11) f_{\mathsf{B}} + \cdots\cdots\cdots + \cdots\cdots \\ & + \cdots\cdots\cdots + \cdots\cdots\cdots + \cdots\cdots\cdots + \cdots\cdots\cdots \\ & + \cdots\cdots\cdots + \cdots\cdots\cdots + \neg\mathsf{s}_1^{\mathsf{A}}\neg\mathsf{t}_1^{\mathsf{A}} \ldots \neg\mathsf{s}_{n_{\mathsf{A}}}^{\mathsf{A}}\neg\mathsf{t}_{n_{\mathsf{A}}}^{\mathsf{A}} f_{\mathsf{A}}(00\ldots00) f_{\mathsf{B}} \end{aligned}$$

where $\vec{b}_1 := \mathcal{E}_{\mathsf{A}}(r,c)$ and thus $f_{\mathsf{A}}(\vec{b}_1) = a_{r,c}$ (cf. Def. 3.13) holds. From Eq. A.1 (p. 162) and the fact that $\mathsf{B} \equiv_\pi B$, it follows immediately that $\mathsf{A} \cdot \mathsf{B} \equiv_\pi A \otimes B$.

The case $\mathsf{s}_k^{\mathsf{B}} <_\pi \mathsf{s}_i^{\mathsf{A}}$ is symmetric.

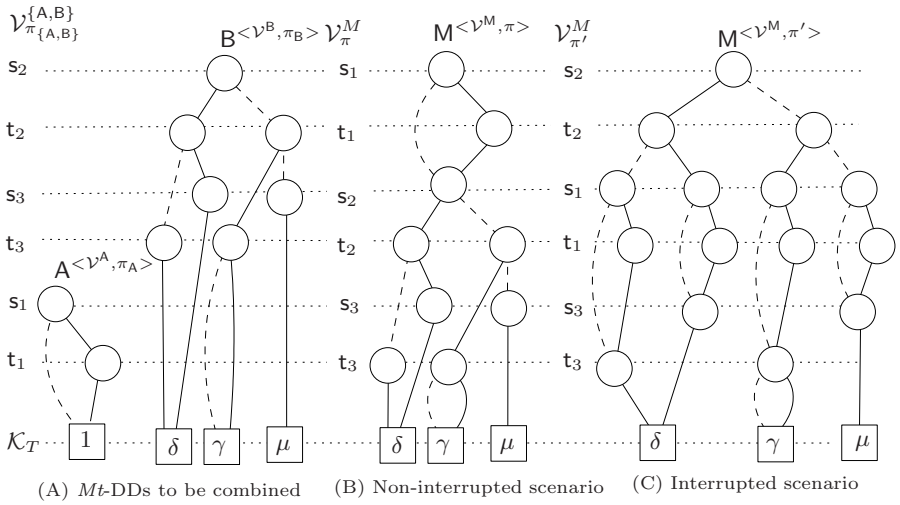


Figure 3.14: Cross-product building and variable orderings ($\mathsf{M} := \mathsf{A} \times \mathsf{B}$)

Interrupted order of the variables
In case of an interrupted order of the variables of A and B, the bits of $\vec{b}_1$ and $\vec{b}_2$ are somehow mixed when combined to build $\vec{b}$. Consequently the co-factor as obtained after making the first $\dim(\vec{b}_1)$ bits constant, may in general not represent matrix B, since $f^M_{\vec{b}_1}(\vec{b}_2) \neq \texttt{Satisfy}(\vec{b}_2, \texttt{getRoot}(\mathsf{B}), \vec{\mathsf{v}}^{\mathsf{B}})$ holds. I.e. due to the overall order π, the element-to-block-wise access pattern while traversing M does not follow the element-to-block-wise combination $A(\mathcal{E}^{-1}(\vec{b}_1)) \cdot B(\mathcal{E}^{-1}(\vec{b}_2))$ as it is mandatory for the encoding the *Kronecker*-product of A and B (cf. Eq. A.1, p. 162). It follows immediately that $\mathsf{A} \cdot \mathsf{B} \not\equiv_\pi A \otimes B$. The case $B \otimes A$ is symmetric. ■

In the following we will employ the symbol $\times$ for emphasizing cross-product building as occurring in case of multiplying DDs with disjoint sets of function variables. In case their sets of function variables are not disjoint we will maintain the usage of the symbol $\cdot$ for referring to the conjunction or multiplication of the DDs. However, one may note that on the level of the DD-algorithms always the same `op`-functions are employed, e.g. `ZAnd` and `ZTimes` in case of *p*ZDDs.

For exemplification and giving a counter example to $\mathsf{A} \times \mathsf{B} \equiv_\pi A \otimes B$, one may refer to Fig. 3.14. On the left (Fig. 3.14.A) the *Mt*-DDs A and B as already employed in previous examples are given. In Fig. 3.14.B and 3.14.C the *Mt*-DD M as resulting from $\mathsf{A} \times \mathsf{B}$ for two different variable orderings π and π', is depicted. Here π induces once a non-interrupted ordering of the variables of A and B (Fig. 3.14.B), whereas π' yields an interrupted one (Fig. 3.14.C). As one can see, the obtained results for M differ, where only $\mathsf{M}{<}\mathcal{V}^{\mathsf{M}}, \pi, \mathbb{D}{>}$ represents the Kronecker product of matrix A and B, $\mathsf{M}{<}\mathcal{V}^{\mathsf{M}}, \pi', \mathbb{D}{>}$ obviously presents a permutation of $A \otimes B$.

If one faces now a situation as illustrated in Sec. 2.5.6, where one needs to build the sum over a set of cross-products of submodel-local transition rate matrices and identity structures (cf. Eq. 2.20, p. 28), the definition of a variable order π, which preserves the non-interrupted order for all variables, may not be possible. It might appear that π destroys the non-interrupted order for one pair of *Mt*-DDs, whereas it produces this order for another pair of *Mt*-DDs. Thus, due to the above considerations, it is clear that a *KO* driven composition scheme is not applicable here, which leads to the following theorem:

Theorem 3.7: *Within a shared BDD-environment, where a fixed variable order π is present and for an index set I. The equation*

$$\sum_{i \in I} \mathsf{A}_i \times \mathbb{1}_i \equiv_\pi \bigoplus_{i \in I} A_i,$$

where $\mathbb{1}_i^j$ are identity matrices of appropriate dimensions, does not hold in general.

Proof: The proof of the above theorem will be achieved by constructing a counter example. To do so, one may consider the following setting:

- $\mathsf{N}_a \equiv_\pi N_a$, and $\mathcal{V}^{\mathsf{N}_a} := \{\mathsf{a}_1, \mathsf{a}_2, \mathsf{b}_2\}$ and
- $\mathsf{N}_b \equiv_\pi N_b$, where $\mathcal{V}^{\mathsf{N}_b} := \{\mathsf{b}_1, \mathsf{b}_2\}$ and
- $\mathsf{N}_c \equiv_\pi N_c$, where $\mathcal{V}^{\mathsf{N}_c} := \{\mathsf{c}_1, \mathsf{a}_2, \mathsf{b}_2\}$ and
- let $\mathbb{1}_a := \mathbb{1}\!<\!\mathsf{c}_1, \mathsf{b}_1\!>$,$\mathbb{1}_b := \mathbb{1}\!<\!\mathsf{c}_1, \mathsf{a}_1, \mathsf{a}_2\!>$ and $\mathbb{1}_c := \mathbb{1}\!<\!\mathsf{a}_1, \mathsf{b}_1\!>$ be the identity structures over the respective variables.

One can easily construct now the sum over the cross-products as follows:

$$\mathsf{M} := \sum_{i \in \{a,b,c\}} \mathsf{N}_i \times \mathbb{1}_i$$

where the variable ordering π is irrelevant for the construction as long as an `Apply`-like algorithm –or its *p*`ZApply` based variants–, for combining the symbolic matrix representations is employed. However, according to lemma 3.6 and for the variable ordering:

$$\mathsf{a}_1 <_\pi \mathsf{a}_2 <_\pi \mathsf{b}_2 <_\pi \mathsf{c}_1 <_\pi \mathsf{b}_1$$

it follows immediately

$$p\texttt{ZAnd}(\times, \texttt{getRoot}(\mathsf{N}_b), \texttt{getRoot}(\mathbb{1}_b)) \not\equiv \mathbb{1}_b \otimes \mathsf{N}_b$$

A re-definition of π, so that the above inequality does not hold, destroys the non-interrupt variable ordering for A. Thus within the above example we have:

$$\nexists \pi : \sum_{i \in \{a,b,c\}} \mathsf{N}_i \times \mathbb{1}_i \equiv_\pi \bigoplus_{i \in \{a,b,c\}} \mathsf{N}_i$$

■

According to the above discussion, it is clear, that situations exist, where the `Apply`-algo. or its *p*`ZApply` based variants are still capable of cross-product building, but where a Kronecker product (*KP*) delivers a wrong result. This result is captured in the following definition.

Definition 3.15: Kronecker operator compliant structure (*KO compliant*)

A high-level model M consisting of n submodels S_i, each with its own set of variables ($\mathfrak{S}_i := \{\mathfrak{s}_1^i, \ldots, \mathfrak{s}_{k_i}^i\}$) possesses a *KO compliant* structure if

$$\exists \pi : \mathfrak{s}_1^1 <_\pi \cdots <_\pi \mathfrak{s}_{k_1}^1 <_\pi \mathfrak{s}_1^2 <_\pi \cdots <_\pi \cdots \mathfrak{s}_{k_{n-1}}^{n-1} <_\pi \mathfrak{s}_1^n <_\pi \cdots <_\pi \mathfrak{s}_{k_n}^n .$$

In case such a order π does not exists, M is said to be not *KO compliant*.

The above theorem directly gives that high-level models, which do not possess *KO compliant* partitionings, can not be analyzed by applying a *Sync* driven decomposition strategy and a *KO* driven composition scheme for obtaining the models potential SG (cf. Sec. 2.5.3, p. 28). –Alternatively one could compute $\tilde{N}_i := \mathbb{1}_i^1 \otimes N_i \otimes \mathbb{1}_i^2$ and apply a respective permutation matrix on $\tilde{N}_i$ ($N_i = P\tilde{N}_i$), so that $\sum_{i \in \{a,b,c\}} N_i$ delivers the desired result. This strategy is problematic, in case where M is not stored explicitly but generated on-the-fly, so that a matrix entry $M(i,j)$ is computed when needed by applying the respective *KO* driven scheme on the local matrices (N_i).

We will come back to this discussion, when contemporary symbolic SG generation techniques are investigated thoroughly (cf. Sec. 4.7, p. 108ff).

Limitations of "pure" symbolic solvers

In the scope of performability analysis one calculates transient- or steady state probabilities by making use of iterative methods, which heavily employ matrix-vector-multiplication (cf. Sec. 2.2.2, p. 13ff). Pure symbolic solvers employ then not only a symbolic matrix representation, they also make use of symbolic vector representations. Such a symbolic representation, rooted in a node n is obtained analogously to Def. 3.11, where each probability p for a state $\vec{s}$ is stored within a terminal node, so that $\texttt{Satisfy}(\mathcal{E}(\vec{s}), n, \vec{\mathsf{s}}) = p$ holds. In this context ZDD based matrix- and / or ZDD based vector representation as illustrated in Fig. 3.11 is hampered by the following conditions:

(1) Pure symbolic matrix-matrix or matrix-vector multiplication is known to be not very efficient, due to the multitudinous ZDD-traversals before accessing the values of the matrix- or vector elements themselves.

(2) The elements of the solution / iteration vector(s) may be all different, thus its symbolic representation may not be compact at all.

(3) The elements of diagonal of the generator matrix, which are the negative row sums of the transition rate matrix, may be all different. Thus their integration within the symbolic matrix representation may lead to a significant blow up in the number of nodes of the latter.

(4) The iterative solution methods require the manipulation of the vector, thus extensive restructuring of the symbolically represented vector occurs.

(5) Some iterative solution methods require an ordered row- or column-wise access to the matrix elements. As pointed out in Sec. 3.5.2 under an interleaved ordering of row - and column indices a simple dfs-traversal can not deliver such a sequence. Instead one would require to extract a column- or row by BDD-manipulation and subsequently extract the single entries of the obtained *Mt*-DD. Consequently such a procedure leads to an overhead as induced by the additionally required and probably extensively executed ZDD based operations.

These considerations have lead to the development of hybrid solvers, as introduced in the context of ADDs in [Par02], where this concepts will be adapted to ZDDs now.

3.5.3 Extending ZDDs for efficiently computing matrix-vector products

The iterative solvers as developed in [Par02] employ a hybrid approach in which the transition rate matrix is stored symbolically by the means of an ADD and the (iteration) vectors are stored as arrays. In the following sections we will briefly introduce the basic ideas of the hybrid approach, but adapted to ZDD based matrix representation.

Index-labeled ZDDs for matrix representation

If n Boolean variables are used for state encoding, there are 2^n potential states of which only a small fraction may be reachable. For exemplification one may refer to Fig. 3.11. The third row and third column of matrix A as given in Fig. 3.11.iv (p. 60) is filled with zeroes only. This stems from the circumstance that the binary label (10) is the label of a non-reachable state ($(10) \notin \mathbb{S}$). Thus all paths containing state label (10), either encoded within the s- or t-variables end-up in the terminal *0-node*, so that the respective matrix positions are filled with zeroes. Allocating entries for unreachable states within the vectors is a waste of memory space and severely restricts the applicability of the algorithms, e.g. if storing probabilities as doubles, a vector with about 134 million entries will already require 1 GByte of RAM. Therefore a dense enumeration scheme for the reachable states has to be implemented. The indexing scheme as introduced for ADDs in [Par02] is adaptable to the case of ZDDs, so that: (a) a dense indexing of reachable states is realized, so that the processing / storing

of *dummy*-entries, i.e. entries referring to non-reachable states, within the solution/iteration vector is avoided. (b) Row and column index of a matrix entry $m_{r,c}$ are computed while traversing the symbolic representation in a dfs-style.

The set of assignments fulfilling the boolean function as defined by a BDD can be indexed by a dense enumeration scheme. The main idea is now to extend BDDs in such a way, that during dfs traversal one computes the index of the assignment as induced by the path currently traversed. However, this of course requires an 1:1 relation between paths and assignments.[2] For realizing the indexing of fulfilling assignments, each BDD node is equipped with an *offset*. Such an *offset* of a node n is hereby defined as the number of assignments fulfilling the boolean function rooted in $\texttt{else}(n)$ ($|Sat_{\texttt{else}(n)}|$), where for $\texttt{else}(n) = \textit{1-node}$ this number is 1 and for $\texttt{else}(n) = \textit{0-node}$ it is 0. *Sat* is the function as defined in Eq. 3.1 in combination with the respective version of algorithm `Satisfy` (cf. Algo. B.2, p. 164). If one descends now the graph rooted in node n, but by following the `then`-edge, it is clear that $\texttt{Offset}(\texttt{else}(n))$ binary labels with a smaller index exist. In order to determine now the index of an assignment $\vec{b}$ as induced by a path p one simply needs to sum up the offsets at nodes left through their out-going `then`-edge. Consequently the *0-sup.* reduction is not problematic since the `then`-edge of a respective node leads directly to the terminal *0-node*, so that its offset will never be considered. The adapted algorithm for *offset*-labeling *z*-BDDs is omitted here, the interested reader may refer to [Zim05] for details.

In contrast to the author of [Par02] we speak of *index-labeling* when it comes to the number-labeling scheme of symbolic matrix representations and speak of offset-labeling in case of the symbolic representation of the binary encoded set of reachable states. This is justified, since the terminology offset-labeling is in our opinion related to path counting. In the context of symbolic "index"-labeled symbolic matrix representations, a node's "index" does not refer to the number of paths concerning the symbolic representation itself, it refers to the offset, i.e. number of minterms as encoded by the "partner node" found within the symbolic structure representing the set of reachable states. I.e. the nodes of a symbolic represented matrix are not labeled by directly employing the offset-labeling scheme as illustrated above, thus the denotation index-labeling seems to be much more appropriated. The procedure of index-labeling a ZDD based matrix representation can be described as follows: Each non-terminal node of the ZDD `M` representing a real-valued matrix is labeled by the *offset* of the corresponding node within the *z*-BDD `S` representing the set of column, row-indices respectively. Two nodes are hereby understood as being corresponding, if they encode the same i'th bit position and if they belong to paths encoding the same binary encoded colum - or row index respectively. While traversing now the ZDD from top to bottom the indices of the matrix entries are calculated by summing up the index-labels of all nodes which are left via their outgoing `then`-edge. Since the `s` and `t` variables are ordered in an interleaved fashion, row and column indices are computed in an alternated fashion. I.e. the index-labels of nodes labeled with an `s` variable may contribute to the (dense) row index, whereas index-labels of nodes labeled with a `t`-variable may contribute to the column index. This procedure allows one the mapping of each bit-string addressing a row- and column index of a matrix entry, to a pair of natural numbers, rather than interpreting the bit-string itself as the binary encoded column - or row index.
Within BDDs and derivatives thereof isomorphic sub-structures are merged. It might appear now that some reachable states share the same subsequence of binary encoded state labels, but are encoded on different paths within the BDD representing the set of reachable states. However, in case of the *Mt*-DDs representing the transition matrix, it is possible that these identical subsequences are represented by a single path. I.e. within the BDD representing the set of reachable states one has more than one node which correspond to the single node within the *Mt*-DD representing the transition matrix. If the offset-labels of the nodes

[2] Dnc-free BDDs can not be employed here directly, since on each path each skipped variable doubles the number of encoded assignments. In order to avoid this ambiguity, [Par02] suggested to re-insert *dnc*-nodes as long as the edge to be re-directed does not lead to the terminal *0-node*.

contained within the BDD are different an offset collision will occur, i.e. while executing the index-labeling one visits a node carrying a label which differ from the one it should carry with respect to the current recursion. For resolving this situation [Par02] simply duplicates the respective nodes, so that isomorphic nodes are only merged if they are equivalent under Def. 3.2 (p. 34) and also their index-labels match.

As already pointed out above for the offset-labeling of BDDs, the adaptation of the *index*-labeling scheme for ADDs [Par02] to the case of ZDDs does not require the additional allocation of nodes as in case of ADDs. The maintenance of the *0-sup.* reduction rule is hereby based on the fact, that under ZDDs one still has a 1:1 relation between pair of indices and paths. However, the skipping of levels concerning the *index*-labeling algorithm is intricate, since it might occur that a node does not possess a partner node in the *z*-BDD representing the set of column - or row indices. But fortunately it shows that one simply needs to continously iterate on the respective partner *z*-BDD until reaching the level of the current ZDD-node to be index-labeled. Furthermore the case that a node within M does not have a partner node within the *offset*-labeled counterpart will never occur, due to the *0-sup.* reduction rule itself. The interested reader may refer to [Zim05], where further details and the algorithms can be found.

Hybrid *index*-ZDD based representation of matrices

Comparing ZDD based representation of real-valued matrices with conventional sparse matrix techniques yields that the symbolic technique realizes memory space advantages. However, this efficiency comes at the cost of additional computational overhead. This overhead stems from the recursive calls of the routines traversing the ZDDs in order to access the terminal nodes, i.e. the matrix entries. Since the transition rate matrices as stemming from high-level model descriptions are most likely to be block-structured, where a block- or sub-matrix might appear various times, the author of [Par02] concluded that it is a good idea to store the blocks (sub-matrices) in a conventional sparse matrix layout. The decision at which ZDD variable one switches from a symbolic representation to a sparse matrix layout depends hereby on the memory space available, where the respective variable will be denoted as *sparse* variable in the following. In contrast to *index*-labeled ADDs, the root node of the block-matrix to be converted into sparse format might not be the same for all paths, due to the skipping of levels. However, this is not problematic, since the index of nodes eliminated under the *0-sup.* rule would not be taken into account anyway. Experiments carried out showed, that the adjustment of the *sparse* variable to the s-variable at position $\lceil\sim \lfloor \frac{1}{3}|\mathcal{V}_{\mathsf{s}}|\rfloor\rceil$ seems to be a good choice, where [Par02] favored the s-variable at position $\lfloor \frac{2}{3}|\mathcal{V}_{\mathsf{s}}|\rfloor$. So instead of removing the lower third of the DDs only, as suggested in [Par02], the experiments of [Zim05] suggested a removal of two thirds of the variables of the DD representing the respective transition rate matrix.

Block-structured hybrid *index*-ZDD based matrix representations

The interleaved ordering of the s- and t-variables does not allow a column- or row-wise access to the matrix entries when executing a dfs-traversal (cf. Sec. 3.5.2). However, some numerical-solution methods require such (ordered) access patterns, e.g. the *GS*-method (cf. Sec. 2.2.2, p. 14). In order to make use of the good convergence behavior of the *GS*-method, the author of [Par02] developed the *pseudo Gauss-Seidel* solution method (*pGS*). Its main idea is to access blocks of the transition rate matrix in an ordered sequence, e.g. row-wise, whereas the elements within the blocks are visited in an arbitrary order. This realizes *GS* on the level of blocks, whereas within the block-matrices the *JAC*-method must be employed. Consequently such an approach requires ordered access to the blocks, as well as an additional iteration vector, where the size of the latter is determined by the size of the largest block matrix. In contrast the *JAC* and *POW*-method do not need an ordered access, but an iteration vector of the size of the full state probability vector.

For block-structuring the overall transition rate matrix, one simply removes the first b pairs of s- and t-variables of the ZDD based representation. Per each increment of b the number of blocks increases by factor 4, i.e. one obtains $n_{blck} := 2^{2b}$ blocks for b pair of s- and t-variables. After removing the first b s- and the first b t-variables, one ends up with a number of *hybrid* and *index*-labeledZDDs, where each of them represent a different sub-matrix $M_{i,j}$ as contained in M. As more levels are removed, as more blocks one obtains and as faster the solution can be obtained, due to the better convergence behavior of the *GS*-method. But this speed-up comes with an additional memory overhead, since the different block matrices must be administered. Consequently the adequate choice of b is also an optimization question as the determination of the *sparse-variable* was before. In the following the last t-variable, which is removed for constructing the block-structured matrix, is denoted as *block-variable*. For administering the root nodes of the block-matrices. the ADD based approach as developed in [Par02], makes uses of a sparse matrix storage scheme (column-major scheme). Due to the imposed memory size, this limits the applicability of the approach, with respect to the choice of b. Based on this observations and contrary to the ADD based approach of [Par02], the ZDD based solvers as developed in the context of this work, make use of a linked list, rather than a sparse matrix storage scheme, where only non-0-blocks are stored. Doing so enables larger choices of b, where it turned out, that a good choices for $2b$, i.e. the number of variables being removed, lies between a third and a half of all variables the ZDD $\mathsf{M} <\vec{\mathsf{s}}, \vec{\mathsf{t}}>$ is defined on. In contrast [Par02] suggest $2b$ to be a third of the number of all s- and t-variables.

3.5.4 Beyond DD based matrix representations

As already reported above, finding appropriated *block* - and *sparse* variables is an optimization problem, which is up to now answered by practical experience. Given that the ZDD-traversal increases the time per iteration, and that the applicability of the pure *GS*-method is limited to the level of blocks, –the individual blocks can only be accessed via ZDD-traversal,– one may wonder if the intermediate ZDD-levels can not be omitted at all. Our implementation described in [Zim05] automatically assigned *block*- and *sparse*-level to the same variable, if the user-defined choices of them are not consistent. As it turned out, this scheme requires the largest memory space but delivers best run-times per numerical iteration. However, [Zim05] was not the first to develop such an idea. The author of [Meh04], developed a two-fold sparse matrix format, where a sparse-matrix format administers the individual block-entries and where the blocks are also stored in a sparse-matrix format. But nevertheless such an approach, no matter if the block-entries are stored as linked list or under a sparse-matrix format, goes beyond symbolic matrix representation. Therefore a deeper discussion is avoided. This is also justified, since the author of [Meh04] focused on hard-drive swapping strategies, i.e. so called *out-of core* methods for computing state probabilities of CTMCs, where this work focuses on pure RAM based methods only.

3.6 Related work and own contributions

In order to obtain a picture of the field of BDD based data types and for gaining an overview over the innovations presented in this chapter, one may turn to Fig. 3.15. There from top to bottom the different refinements as suggested by different authors can be tracked, the contributions of this work are given bold-faced. The different innovations will be discussed now in detail.

Binary Decision Diagrams (BDDs) and derivatives thereof are widely used in today's CAD-tools, since they are known to be extremely efficient in the representation of boolean functions. Their basic form, Binary Decision Trees are based on the observation that the n variables of a boolean function can be recursively replaced by the constant 1 or 0. This expansion of boolean functions is commonly denoted as *Shannon expansion*. In 1938 the American mathematician Shannon published a paper about symbolic representation of switching functions [Sha00]. I.e. he showed how switching functions can be mapped via expansion onto

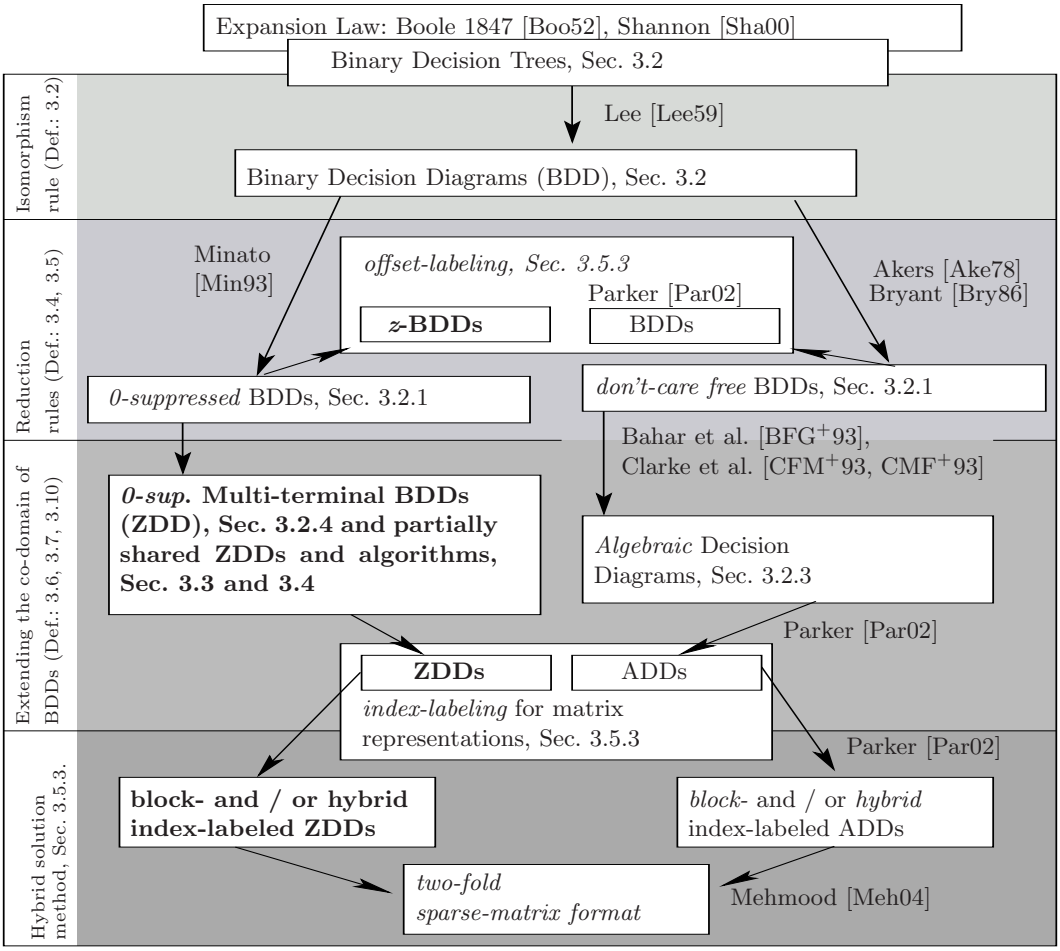


Figure 3.15: Development and inheritance of concepts within BDT based data types

a boolean algebra, and thus laid the basics of modern hardware verification. Consequently the term symbolic representation of boolean functions or symbolic set representation itself, as it is used nowadays, is not very precise. One should better speak of graph based representation and manipulation of discrete (boolean) functions, when it comes to their BDD based representation, which is also the appellation Bryant employed in his ground-breaking paper [Bry86]. But due to the broad acceptance of this terminology we maintain it here.
Based on the work of Shannon, Lee [Lee59] developed the concept of binary decision programs, which can be viewed as variants of BDDs. In 1978 Akers [Ake78] introduced the *dnc*-reduction rule, as well as the possibility of employing inverted arcs between the BDD nodes. But it was up to Bryant [Bry86] to define an ordering relation on the boolean variables and thus to introduce reduced **ordered** BDDs, allowing him to develop algorithms for the efficient manipulation of BDDs, where the `Apply`-algorithm is the most important one. This generic algorithm follows the Shannon expansion and allows one to efficiently apply a binary operator directly onto BDDs, where a new BDD representing the resulting function is generated. Its development was the breakthrough for the success of BDDs in many of today's applications, since the efficiency of BDDs and derivatives thereof was from now on not only restricted to memory savings. This development inspired many researchers, so that nowadays a broad range of graph based data types and algorithms for their efficient manipulation exists. The research activities of the past decades have hereby led also to other forms of expanding boolean functions, suiting all different kinds of applications. For an overview the interested reader may refer to [MT98] and [DB98].

Since one may interpret the boolean function represented by a BDD also as a characteristic function of a set, the elements of which are labeled by binary strings, the relation to efficient set representation and manipulation is evident. Consequently it is not surprising that BDDs are the most prominent representative when it comes to symbolic state space or state graph representation. In the field of performability modeling the most prominent BDD based data structures are *multi-terminal* - or *algebraic decision diagrams* [CFM$^+$93, CMF$^+$93, BFG$^+$93], and their offset-labeled variants [Par02]. Another branch contains *multi-valued decision diagrams* (MDDs) [KVBSV98] and *matrix diagrams* (MxDs) [Min01]. The interested reader may find a detailed survey in [MP04], where binary and also non-binary decision diagrams in the context of performability modeling are discussed.

Boolean functions with sparsely enumerated satisfaction sets, i.e. functions where the fulfilling assignments possess many 0-assigned bit positions, may yield large BDD based representations. In such settings *zero-suppressed* BDDs (*z*-BDDs) [Min93] have shown to be very helpful. Thus it is a very natural way of replacing the *don't-care*-reduction rule in case of ADDs with the *zero-suppressing* reduction rule, as it was first suggested by us in [LS03b, LS06a]. As simple as the idea of extending *z*-BDDs to the multi-terminal case sounds, as difficult the efficient algorithmic manipulation of the resulting ZDDs in shared BDD environments turned out to be. These difficulties are closely related to the fact that in the presence of the *0-sup.*-reduction rule, the set of boolean variables a *Mt*-DD is defined over matters. This circumstance forced us to develop the concept of *partially shared* ZDDs (*p*ZDDs), i.e. to consider ZDDs with differing sets of boolean variables. This new *Mt*-DD based data type required a re-design of Bryant's BDD-algorithms [Bry86], yielding most importantly the the *p*`ZApply`-, *p*`ZAnd`-, *p*`ZRestrict`- and *p*`ZAbstract`-algorithm.

The concept of offset and index-labeling in case of *z*-BDDs and ZDDs as introduced in this chapter, which is a precondition for the efficient BDD based solution of CTMCs, are adaptations of the approaches developed in [Par02] for BDDs and ADDs. Using ZDDs we had hereby to adapt the concept of offset- or index-labeling: With ADDs, skipped variables lead to a re-insertion of nodes, whereas for ZDDs, skipped nodes correspond to zero-valued variables for which the offset is irrelevant. Therefore, in the ZDD case, skipped nodes do not have to be reinserted, which keeps the symbolic data structure in general more compact.
Analogously to the author of [Par02] we also adapted the idea of extending index-labeled ZDDs by making use of sparse matrix formats, yielding block-structured and / or hybrid *index*-ZDDs. However, in contrast to [Par02] we discovered that a linked list for administering the non-zero block-matrices is favorable, allowing us to pick an arbitrary value for the *block*-variable. Given a flexible choice of *block* and *sparse* variable at hand, we discovered, that the removal of all ZDD-variables residing between linked list for storing block-entries and spare-matrix formats for individual block-entries delivers best iteration numbers and times and the cost of a strongly increased memory consumption. A similar approach was also suggested in [Meh04], however in contrast to this work, the author of [Meh04] focused on *out-of core* methods, for computing state probabilities of CTMCs.

Several case studies have provided evidence that the ZDDs are superior to ADDs when representing Markov chains which are derived from typical high-level specifications (cf. Sec. 2.3). It has been found that, in general, the ZDD based representation is more compact than the ADD based representation. This has the positive effect that the construction and manipulation of the symbolic representations, as well as the times for computing numerical solutions are also reduced (cf. Chapter 5, p. 119ff).

4

The Activity/Reward-local Scheme: Symbolic SG based Analysis of High-level Markov Reward Models

At its core the activity/reward-local approach applies a selective breadth-first-search scheme for explicit SG exploration, a scheme for symbolic composing state graphs, algorithms for symbolic reachability analysis, and a scheme for generating symbolic representations of reward functions. This yields a symbolic representation of the high-level Markov reward model's underlying reward-annotated state-transition system. Based on the obtained symbolic representation a probability distribution on the set of system states is computed, where the hybrid solution method, as already discussed in the previous chapter, is applied. Algorithms based on graph-traversal allows one then to efficiently compute the specified measures modeling the performability of the system under study. Since an important goal can be seen in the reduction of the number of system states, the state probability of which must be computed, the basic activity/reward-local scheme will be extended to cope with models containing submodel-imposed symmetries, where a symbolic algorithm for the state graph reduction will be presented.

4.1 Organization of the chapter

The activity/reward-local scheme consists of two aspects: (a) the generation procedures for obtaining a symbolic representation of a high-level model's underlying *esLTS*, denoted as activity-local scheme and (b) the procedure for generating symbolic representations of reward functions, addressed as reward-local scheme. Before introducing these schemes we need to introduce the model world, which is done in Sec. 4.2. The discussion ranges from helpful definitions up to the limitations of the high-level models to be handled. This is necessary, since the activity/reward-local scheme is not limited to a single high-level model description method.

Sec. 4.3 explains the basics of the activity/reward-local scheme for efficiently constructing symbolic SG representations. In order to keep the discussion simple we first restrict ourself to purely Markovian high-level models[1] and the basic SG generation and encoding scheme. The completeness and correctness of the basic scheme is covert in Sec. 4.4. Sec. 4.5 discusses how symbolic representations of reward functions and the distribution of state probabilities on the set of reachable states can be obtained. Given symbolically represented reward functions and the vector of state probabilities, one is enabled to determine moments of user-defined performability measures, via a new graph-traversing algorithm as presented at the end of section 4.5.

Now the discussion is ready for Sec. 4.6, where useful extensions of the basic scheme are introduced. These extensions range from the application of the lumping theorem in case of explicitly modeled symmetries, up to an extended activity/reward-local scheme for handling high-level models containing prioritized immediate activities.

Since the last years have seen tremendous efforts on the sector of symbolic representations for Markov models, a detailed discussion of related work is appropriate, which is done in Sec. 4.7. The chapter is concluded by Sec. 4.8, which indicates our pre-published material.

[1] Models where the set of immediate activities is empty

4.2 Model world

The model world as to be developed in this section does not contain the explicit definition of a partitioning of the high-level models. Since contrary to other compositional SG generation procedures, the scheme to be developed applies an automatized activity-wise model decomposition. To do so, the only model structure required is an arbitrary connection relation defined on the set of state variables and activities, each high-level model consists of. However, as a consequence of this, compositionally constructed high-level models, where the *Sync* composition method is employed, must be modified in such a way that the activities to be jointly executed among the submodels are merged posteriori to SG generation, which we already described in Sec. 2.4 (p. 24ff).

4.2.1 Static properties

(1) A high-level model M consists of a finite ordered set of discrete state variables (SVs) $\mathfrak{s}_i \in \mathfrak{S}$, and a finite set of activities $l_i \in \mathcal{A}ct$, where the sets of SVs and activities are assumed to be disjoint ($\mathfrak{S} \cap \mathcal{A}ct = \emptyset$).

(2) SVs and activities are somehow related, which is reflected by the following connection relation $\mathcal{C}on$:

$$\mathcal{C}on \subseteq (\mathfrak{S} \times \mathcal{A}ct) \cup (\mathcal{A}ct \times \mathfrak{S}). \tag{4.1}$$

Connection relation, SVs and activities allows one to define the following static net structure:

Definition 4.1: Simple SV-Activity net (`simple` SA net)

A `simple` SA net N is a triple $N := (\mathfrak{S}, \mathcal{A}ct, \mathcal{C}on)$, where

- $\mathfrak{S}$ is the ordered finite tuple of SV $(\mathfrak{s}_1, \ldots \mathfrak{s}_{|\mathfrak{S}|})$,
- $\mathcal{A}ct$ is the ordered finite tuple of activities $(l_1, \ldots l_{n_{\mathcal{A}ct}})$, with $n_{\mathcal{A}ct} := |\mathcal{A}ct|$ and $\mathfrak{S} \cap \mathcal{A}ct = \emptyset$,
- $\mathcal{C}on$ is the connection relation as defined in Eq. 4.1.

Throughout the rest of this work, it is assumed that not only SVs steering the execution of an activity l appear in the above connection relation. It is essential that also all SVs influencing the execution weight, - rate or - priority do so. Therefore we define an activity's set of dependent SVs as follows:

Definition 4.2: Set of activity-(in)dependent SVs

Within a `simple` SA net each activity l is connected via $\mathcal{C}on$ with a set of SVs. Such a set of connected SVs shall be denoted as l's set of dependent SVs and it is defined as:

$$\mathfrak{S}_l^{\mathcal{D}} := \{\mathfrak{s}_i \mid \mathfrak{s}_i \in \mathfrak{S} : (\mathfrak{s}_i, l) \in \mathcal{C}on \vee (l, \mathfrak{s}_i) \in \mathcal{C}on\}.$$

The set $\mathfrak{S}_l^{\mathcal{I}} := \mathfrak{S} \setminus \mathfrak{S}_l^{\mathcal{D}}$ is denoted as activity l's set of independent SVs.

All functions steering the behavior of an activity l, are assumed to take only dependent SVs as their input parameters, where each time activity l is executed, its *dependent* SVs $\mathfrak{S}_l^{\mathcal{D}}$ may change, and where the SVs belonging to l's set of *independent* SVs ($\mathfrak{S}_l^{\mathcal{I}}$) maintain their values!

Common Markovian model description methods allow one to employ either (a) *immediate* activities $l \in \mathcal{A}ct^i$ and (b) *timed* activities $l \in \mathcal{A}ct^m$ (cf. Sec. 2.3). Therefore we extend the concept of `simple` SA nets as follows:

Definition 4.3: SV-Activity net (`SA net`)

A `SA net` N is a `simple` SA net $N := (\mathfrak{S}, \mathcal{A}ct, \mathcal{C}on)$, where $\mathcal{A}ct := \mathcal{A}ct^m \cup \mathcal{A}ct^i$, so that

- $\mathcal{A}ct^m$ is the set of timed activities and
- $\mathcal{A}ct^i$ is the set of immediate activities, and
- $\mathcal{A}ct^m \cap \mathcal{A}ct^i = \emptyset$.

The semantics of *immediate* and *Markovian* transition was already discussed in Sec. 2.2.3. The semantics of their inducing high-level counterparts (immediate or timed activities) will be clarified in the following section, when the dynamic behavior of `SA nets` is discussed.

4.2.2 Dynamic properties

Based on the vector-layout for states as introduced in Sec. 2.4 (Eq. 2.17, p. 25) and based on the set of *dependent* (*independent*) SVs for each activity (cf. Def. 4.2) one may define now the following mapping for an activity l and a state $\vec{s}$:

$$\chi_l^{\mathcal{D}}\colon\ \mathbb{N}^{|\mathfrak{S}|} \longrightarrow \mathbb{N}^{|\mathfrak{S}_l^{\mathcal{D}}|} \qquad (\chi_l^{\mathcal{I}}\colon\ \mathbb{N}^{|\mathfrak{S}|} \longrightarrow \mathbb{N}^{|\mathfrak{S}_l^{\mathcal{I}}|}) \tag{4.2}$$

This mapping extracts the values of the *dependent* (*independent*) SVs as contained in a state $\vec{s}$, where for simplicity the shorthand notations $\vec{s}_{D_l} := \chi_l^{\mathcal{D}}(\vec{s})$, and $\vec{s}_{I_l} := \chi_l^{\mathcal{I}}(\vec{s})$ will be employed. The (sub-) vector $\vec{s}_{D_l}$ will be denoted as l's dependent and the (sub-) vector $\vec{s}_{I_l}$ as l's independent state marking concerning state $\vec{s}$.

The model's evolution from state $\vec{s}$ to the next state $\vec{t}$ is achieved by the execution of the activity specific transition or execution functions for each enabled activity. However, before being qualified for defining the enabledness of an activity, one need to define the following:

Definition 4.4: Concession of an activity

An activity-specific predicate function $pred_l : \mathbb{N}^{|\mathfrak{S}_l^{\mathcal{D}}|} \to \{true, false\}$ for a given state $\vec{s}$ and a specific activity $l \in \mathcal{A}ct$ is defined as follows:

$$pred_l(\vec{s}_{D_l}) := x \,\text{where}\, x \in \{true, false\} \Leftrightarrow\ \vec{s} \in \mathbb{S}$$

In case $pred_l(\vec{s}_{D_l}) = true$ one says that activity l has concession in state $\vec{s}$, which is addressed by the notation $\vec{s} \triangleright l$. If $pred_l(\vec{s}_{D_l}) = false$ one writes $\vec{s} \not\triangleright l$.

It is enforced that the SVs, which influence the predicate of an activity l over a state $\vec{s}$ are also elements of $\mathfrak{S}_l^{\mathcal{D}}$.

Having concession alone is not sufficient for an activity l to be executed. In the presence of priorities it is also necessary, that there is no other activity k which possesses concession in $\vec{s}$ and which has a higher priority than l. The priorities not only depend on the activity, they may also depend on the models current state, which is reflected by the following definition:

Definition 4.5: Activity-specific priority function

A priority function $prio_l$ for an `SA net` is a function $prio_l : \mathbb{N}^{|\mathfrak{S}_l^{\mathcal{D}}|} \to \mathbb{N}$ for an activity l and a state $\vec{s}$, which is defined as follows:

$$prio_l(\vec{s}_{D_l}) := x \,\text{where}\, x \in \begin{cases} \mathbb{N}_{>0} \text{ if } l \in \mathcal{A}ct^i \wedge\ \vec{s} \triangleright l \\ \{0\} \ \text{ if } l \in \mathcal{A}ct^m \end{cases}$$

Now we are in the position of defining the enabledness of an activity l as follows:

$$\vec{s}\,[> l \Leftrightarrow \vec{s} \triangleright l \wedge (\nexists k \in \mathcal{A}ct : \vec{s} \triangleright k \wedge prio_k(\vec{s}_{D_k}) > prio_l(\vec{s}_{D_l})) \tag{4.3}$$

where $\vec{s}\,[> l$ means that activity l is enabled. In case l is not enabled, since $\vec{s} \not\triangleright l$ or $\exists k \in \mathcal{A}ct : \vec{s} \triangleright k \wedge prio_k(\vec{s}_{D_k}) > prio_l(\vec{s}_{D_l})$, the notation $\vec{s}\,[\not> l_i$ is employed. This allows one to define the set of enabled activities to be executed in a given state $\vec{s}$:

Definition 4.6: Set of enabled activities

The set of activities enabled in a given state is defined as:

$$\mathcal{A}_{\vec{s}} := \{l \in \mathcal{A}ct \mid \vec{s}\,[> l\}.$$

Each successor state is then obtained by applying the activity-specific transition function δ_l on the source state $\vec{s}$. However, concerning the targeted model world, the model description method specific implementation of δ_l is without interest here. The only thing required is the fact, that δ_l assigns new values only to those positions j of $\vec{s}$, which are associated with SVs of l's set of dependent SVs ($\mathfrak{s}_j \in \mathfrak{S}_l^{\mathcal{D}}$). Positions referring to SVs of l's set of independent SVs ($\mathfrak{s}_j \in \mathfrak{S} \setminus \mathfrak{S}_l^{\mathcal{D}}$) maintain their values. I.e. formally one has:

Definition 4.7: Activity-specific transition function

The activity-specific transition function $\delta_l : \mathbb{N}^{|\mathfrak{S}|} \to \mathbb{N}^{|\mathfrak{S}|}$, which returns the target state $\vec{t}$ for a given source state $\vec{s}$ and an activity l is defined as follows:

$$\delta_l(\vec{s}) := \vec{x} \,\text{where}\, \vec{x}\,[i] := \begin{cases} \vec{t}\,[i] \in \mathbb{N} \Leftrightarrow \mathfrak{s}_i \in \mathfrak{S}_l^{\mathcal{D}} \wedge \vec{s} \in \mathbb{S} \\ \vec{s}\,[i] \in \mathbb{N} \Leftrightarrow \mathfrak{s}_i \in \mathfrak{S}_l^{\mathcal{I}} \wedge \vec{s} \in \mathbb{S} \end{cases}$$

The set of all activity-specific transition functions allows one to define the global transition function of the high-level model:

Definition 4.8: State-to-State transition function on a `SA net`

The state-to-state transition function $\Delta : \{\mathbb{S} \times \mathcal{A}ct\} \to \mathbb{S}$ on a `SA net` is defined as follows:

$$\Delta(\vec{s}, l) := \delta_l(\vec{s}) \Leftrightarrow l \in \mathcal{A}_{\vec{s}}$$

where $\mathcal{A}_{\vec{s}}$ is the set as defined in Def. 4.6. This state-to-state transition function Δ can be generalized to a function $\Delta^* : \mathbb{S} \times \mathcal{A}ct^* \to \mathbb{S}$, which is inductively defined as follows:

$$\begin{aligned} \Delta^*(\vec{s}, \epsilon) &:= \vec{s} \\ \Delta^*(\vec{s}, \omega l) &:= \Delta(\Delta^*(\vec{s}, \omega), l) \end{aligned}$$

where ϵ is the empty word.

The (global) transition function Δ, together with the initial state $\vec{s}^{\,\epsilon}$, which assigns an initial value to each SVs, allows one to generate a set of states $\mathbb{S}$ for the `SA net`:

Definition 4.9: Set of reachable states and set of reachable transitions

The set of reachable states for a `SA net`, a transition function Δ and an initial state $\vec{s}^{\,\epsilon}$ is inductively defined by

(1) $\vec{s}^{\,\epsilon} \in \mathbb{S}$

(2) $(\vec{s} \in \mathbb{S} \wedge \exists l \in \mathcal{A}_{\vec{s}}) \Rightarrow \Delta(\vec{s}, l)) \in \mathbb{S}$

Each transition of a `SA net` with source state $\vec{s}$, activity l and target state $\vec{t} := \Delta(\vec{s}, l)$ is a triple $(\vec{s}, l, \vec{t})$. The set of all such triples yields a *LTS* $T \subseteq \mathbb{S} \times \mathcal{A}ct \times \mathbb{S}$.

An element of the LTS T $(t := (\vec{s}, l, \vec{t}))$ is denoted as reachable if $\exists \omega \in \mathcal{A}ct^*$, so that $\vec{s} = \Delta^*(\vec{s}^{\,\epsilon}, \omega) \in \mathbb{S}$. In case such an activity-execution sequence ω exists not, i.e. $\vec{s}$ is not reachable, t is denoted as non-reachable transition. *In case a LTS contains more than the actually reachable states it is denoted as potential transition system.* With the help of Δ and $\vec{s}^{\,\epsilon}$ one is enabled to construct the set of reachable states $\mathbb{S}$, as well as the labeled transition system T for a given `SA net`. The set of reachable states can be divided into the following partitions:

Definition 4.10: Partitioning the set of reachable states

For the set of reachable states $\mathbb{S}$ of a `SA net`, we define the following partitioning:

$$\mathbb{S} = \mathbb{S}_{Van} \cup \mathbb{S}_{Tan} \cup \mathbb{S}_{Absorb},$$

where

- $\mathbb{S}_{Van}$ is the set of vanishing states: $\mathbb{S}_{Van} : \{\vec{s} \in \mathbb{S} \mid \exists l \in \mathcal{A}ct^i : \vec{s}\,[> l\}$
- $\mathbb{S}_{Tan}$ is the set of tangible states: $\mathbb{S}_{Tan} : \{\vec{s} \in \mathbb{S} \mid \exists l \in \mathcal{A}ct^m : \vec{s}\,[> l\}$
- $\mathbb{S}_{Absorb}$ is the set of absorbing states: $\mathbb{S}_{Absorb} : \{\vec{s} \in \mathbb{S} \mid \nexists l \in \mathcal{A}ct : \vec{s}\,[> l\}$

The semantics of these different types of states was already discussed in Sec. 2.2.3.

A standard interleaving semantics resolves concurrency by executing all enabled activities. Consequently one needs to decide which activity is executed first, i.e. to schedule the activities. This situation is commonly denoted as racing and it needs to be somehow resolved. According to Eq. 4.3 among the activities with concession always the one(s) with the highest priority win(s). Furthermore Def. 4.5 gives that Markovian activities always have the same priority level 0. Consequently analogously to the GSPN semantics, the situation is constructed, that immediate activities granted concession always suppress the enabledness of Markovian activities which are granted concession as well. Due to this dominance of immediate activities over Markovian the race-situation for the two classes can be investigated separately:

Immediate activities

The non-determinism in case of immediate activities is commonly resolved by computing execution probabilities for the immediate activities enabled in a vanishing state. Since the modeler might not wish that these probabilities are uniformingly distributed, common high-level model description methods allow one to equipped each immediate activity with a state-dependent execution weight. The weight returning function of an activity is defined as follows:

Definition 4.11: Activity-specific weight returning function

An activity-specific weight returning function $\Omega_l : \mathbb{N}^{|\mathfrak{S}_l^{\mathcal{P}}|} \times \mathbb{N}^{|\mathfrak{S}_l^{\mathcal{P}}|} \to \mathbb{R}^+$ for $l \in \mathcal{A}ct^i$ over a transition system T of a `SA net` is defined as follows

$$\Omega_l(\vec{s}_{D_l}, \vec{t}_{D_l}) := x \,\text{where}\, x \in \mathbb{R}^+$$

where $\vec{t}_{D_l} := \chi_l^{\mathcal{P}}(\Delta(\vec{s}, l))$ (cf. Def. 4.2). By aggregating the weights of transitions emanating from the same state, one obtains the (global) weight returning function $\Omega : \mathbb{S}_{Van} \to \mathbb{R}^+$:

$$\Omega(\vec{s}) := \sum_{l \in \mathcal{A}_{\vec{s}}} \Omega_l(\vec{s}_{D_l}, \vec{t}_{D_l}).$$

This allows one to calculate the probability for each transition $(\vec{s}, l, \vec{t}) \in T$ as induced by an immediate activity $l \in \mathcal{A}ct^i$ as follows:

$$\Pi(\vec{s}, l, \vec{t}) := \frac{\Omega_l(\vec{s}_{D_l}, \vec{t}_{D_l})}{\Omega(\vec{s})}. \tag{4.4}$$

However, the case $\Omega(\vec{s}) = 0$ needs to be eliminated. This is achieved by simply granting only concession to activities, those execution weight $\neq 0$, this strategy is formalized in Def. 4.13. In such a setting it holds then that the execution probabilities of all activities enabled in a given vanishing source state sums up to 1:

$$\forall \vec{s} \in \mathbb{S}_{Van} : \sum_{l \in \mathcal{A}ct^i \cap \mathcal{A}_{\vec{s}}} \Pi(\vec{s}, l, \Delta(\vec{s}, l)) = 1$$

At this point it is useful to indicate, that the above suggested calculation of execution probabilities requires global knowledge about all immediate activities to be executed in a given state. This turns out to be problematic for a SG generation scheme which employs local knowledge about activities only.

Timed activities

Analogously to a GSPN semantics it is assumed that an enabled timed activity l is executed after a delay, where the latter is sampled from a neg. exponential distribution with parameter λ_l. The transition from one state to another itself is instantaneous and does not consume any time. Analogously to the weight returning function we define:

Definition 4.12: Rate returning function

An activity-specific rate returning function $\Lambda_l : \mathbb{N}^{|\mathfrak{S}_l^{\mathcal{P}}|} \times \mathbb{N}^{|\mathfrak{S}_l^{\mathcal{P}}|} \to \mathbb{R}^+$ for an activity $l \in \mathcal{A}ct^m$ over a transition system T of a `SA net` is defined as follows:

$$\Lambda_l(\vec{s}_{D_l}, \vec{t}_{D_l}) := x \,\text{where}\, x \in \mathbb{R}^+$$

where $\vec{t}_{D_l} := \chi_l^{\mathcal{P}}(\Delta(\vec{s}, l))$ and $\vec{s} \in \mathbb{S}$. By aggregating the rates of transitions between the same pair of states, one obtains the (global) rate returning function $\Lambda : \mathbb{S} \times \mathbb{S} \to \mathbb{R}^+$:

$$\Lambda(\vec{s}, \vec{t}) := \sum_{l \in \mathcal{A}ct^m \cap \mathcal{A}_{\vec{s}}} \Lambda_l(\vec{s}_{D_l}, \vec{t}_{D_l})$$

According to the above definitions it might occur that there are enabled activities having execution weights or rates equal to 0. Such activities should be removed from the set of enabled ones, so that they do not interfere with the enabledness of other activities. In order to guarantee this, one simple needs to re-define the condition for an activity for having

concession:

Definition 4.13: Concession of an activity under a weight or rate returning function

An activity l is said to have concession in a state $\vec{s}$ ($\vec{s} \triangleright l$) if the activity specific predicate function $pred_l$ for a state $\vec{s}$ is true and the activity's execution rate or weight $\neq 0$. I.e. for a reachable state $\vec{s} \in \mathbb{S}$ one has:

$$\vec{s} \triangleright l \Leftrightarrow \left[pred_l(\vec{s}_{D_l}) = true \wedge F_l(\vec{s}, \Delta(\vec{s}, l)) \neq 0\right]$$

where F_l is defined as

$$F_l(\vec{s}, \vec{t}) := \begin{cases} \Omega_l(\vec{s}_{D_l}, \chi_l^{\mathcal{D}}(\Delta(\vec{s}, l))) \text{ if } l \in \mathcal{A}ct^i \\ \Lambda_l(\vec{s}_{D_l}, \chi_l^{\mathcal{D}}(\Delta(\vec{s}, l))) \text{ if } l \in \mathcal{A}ct^m \end{cases}$$

Due to the above extension of the concession rule, an immediate activity having concession but a zero weight, does not suppress the execution of another activity having concession in the very same state and a execution weight or rate other than 0. This is important when it comes to symbolic reachability analysis in the presence of Markovian and immediate activities (as to be discussed in Sec. 4.6.2, p. 106)

The discussion carried out so far, leads now to the definition of generalized stochastic `SA net` *model* (`GS-SA net`) incorporating timed behavior:

Definition 4.14: A Generalized Stochastic `SA net` *model* (`GS-SA net`)

A `GS-SA net` is a 5-tuple $N := (SA, \Delta, \vec{s}^{\,\epsilon}, \Omega, \Lambda)$ where

- N is a `SA net`,
- Δ is the global transition function as specified in Def. 4.8,
- the dedicated state $\vec{s}^{\,\epsilon} \in \mathbb{S}_{Tan}$ gives an initial value for each SV,
- $\Omega : \mathbb{S} \times \mathbb{S} \to \mathbb{R}^+$ is the global weight returning function of Def. 4.11,
- $\Lambda : \mathbb{S} \times \mathbb{S} \to \mathbb{R}^+$ is the global rate returning function of Def. 4.12

In case $\mathcal{A}ct^i = \emptyset$ one speaks of a stochastic `SA net` *model* (`S-SA net`).

Thus it is clear that for a given initial state $\vec{s}^{\,\epsilon}$ a `GS-SA net` can be mapped to a *esLTS* by employing the functions Δ, Ω and Λ on SVs and activities, where a vector layout for each state, and an appropriate connection relation on SVs and activities is present. This setting constitutes the framework or model world for the symbolic SG generation approach to be developed in this work.

Finally, we define the semantics of performance variables, enabling the modeler to define complex performance measures on the level of the high-level models, rather than on the level of their underlying *esLTS*. A performance variable (PV) consists of sets of rate reward - and / or set of impulse reward functions. A rate reward function defines hereby the reward gained by the model in a specific state, to do so a rate reward returning function takes a reachable state as input and maps it to a real number. In contrast an impulse reward defines the reward as obtained by executing a specific activity in a specific state, i.e. the values of impulse reward functions are at least activity-dependent but may also be state-dependent. In both cases the specific reward values are defined to be time independent, thus they can be generated during SG construction (cf. Sec. 2.3.4). Analogously to weight and rate returning functions we also emphasize the principle of locality here, i.e. we assume

a connection relation between a rate or impulse reward and the SVs taken by the reward returning function as input parameters. Thus we define the reward functions to take sub-vectors (!) of the high-level model's states as their input parameter, where in case of impulse rewards this sub-vector is restricted to the set of dependent SVs of the associated impulse generating activity. In case of a rate reward function, this set of SVs can be an arbitrary subsets of the high-level model's SVs. This gives us the following definitions:

Definition 4.15: Rate reward

A rate reward r on a `GS-SA net` is specified by the rate reward returning function $\mathcal{R}_r : \mathbb{N}^{|\mathfrak{S}_r^{\mathcal{D}}|} \rightarrow \mathbb{R}$ and defined as follows:

$$\mathcal{R}_r(\vec{s}_{D_r}) := x \,\text{where}\, x \in \mathbb{R} \Leftrightarrow \vec{s} \in \mathbb{S}_{Tan}$$

where $\mathfrak{S}_r^{\mathcal{D}} \subseteq \mathfrak{S}$ is the set of SVs the evaluation of the reward function depends on, denoted as set of dependent SVs of reward r. Analogously to Eq. 4.2 $\vec{s}_{D_r}$ is the shorthand notation of a mapping of a state $\vec{s} \in \mathbb{S}$ onto the positions associated with r's dependent SVs.

The set of all rate reward functions defined on a given high-level model will be denoted $\mathcal{R}$, so that a set of rate rewards may be obtainable for each state. Due to the computation of rate rewards as introduced in Sec. 2.2.2 (p. 15ff), the above defined rate reward returning function on a `GS-SA net` can be directly employed when aggregating rate rewards, in order to construct complex performance variables. Contrary to this, impulse rewards must be weighted with the respective activity rate before aggregation of multiple impulse reward functions can be done (cf. Eq. 2.9 and 2.12, p. 16). This is needed, since more than one activity may contribute to an impulse reward, so that the individual activity-dependent impulse reward values are required to be aggregated. This makes impulse reward more complex, where the following definition covers this aspect:

Definition 4.16: Impulse reward

An impulse reward j of a `GS-SA net` is received if a specific activity k is executed, and where the activity-specific impulse reward generated by this activity may also be state-dependent. The set of (Markovian) activities which contribute to impulse reward j will be labeled as $\mathcal{A}ct_j^m$ and denoted as j's set of impulse reward inducing activities. The impulse reward returning function $\mathcal{I}^j : \mathbb{S}_{Tan} \rightarrow \mathbb{R}$, for impulse reward j can then be defined as follows:

$$\mathcal{I}^j(\vec{s}\,) := \sum_{k \in \mathcal{A}_{\vec{s}} \cap \mathcal{A}ct_j^m} \mathcal{I}_k^j(\vec{s}_{D_k}) \cdot \Lambda_l(\vec{s}_{D_k}, \delta_k(\vec{s}_{D_k}))$$

where Λ_l is the rate returning function of Def. 4.12 and $\mathcal{I}_k^j(\vec{s}_{D_k})$ is the impulse reward returning function of activity k with respect to impulse reward j and which we restrict to k's set of dependent SVs:

$$\mathcal{I}_k^j(\vec{s}_{D_k}) := x \,\text{where}\, x \in \mathbb{R} \Leftrightarrow k \in \mathcal{A}_{\vec{s}} \cap \mathcal{A}ct_j^m \wedge \vec{s} \in \mathbb{S}_{Tan}$$

Within the above definition $\mathcal{I}_k^j(\vec{s}_{D_k})$ is constructed in such a way that it matches the basic definition of impulse reward functions which we already gave in the context of low-level MRMs (cf. Sec. 2.2.1 (p. 9)).
One simply needs to employ the activity-specific transition function δ_k on a source state $\vec{s}$, so that the pair $(\vec{s}, \delta_k(\vec{s}))$ can serve as input for an impulse reward function, so that $\mathcal{I}_k^j(\vec{s}_{D_k})$ is equivalent to function $\mathcal{I}^j(\vec{s}, \delta_k(\vec{s}))$ as defined in Def. 2.4 (p. 11). In cases where more than one activity contributes to the actual value of an impulse reward, it is important that one

multiplies each activity-dependent reward function $\mathcal{I}_k^j(\vec{s}_{D_k})$ with the respective activity rate before the sum over the activity-dependent impulse reward functions is computed. This is exactly what we did in the above definition, when specifying impulse function $\mathcal{I}^j(\vec{s})$.
For a high-level model there might be more than one impulse reward returning function for an activity k, which of course depends on the user-defined set of impulse rewards. The set of all impulse rewards defined for a given high-level model will be denoted as $\mathcal{I}$

Based on the above definitions one is now enabled to combine rate and impulse rewards to build complex performance variables. If these performance variables are equipped with a notion of time, a respective performability measure can be computed by applying the equations of Sec. 2.2.2 (p. 15ff.). Therefore we assign a type to each PV, so that one is enabled to distinguish between steady state or transient state performability measures. Since the notion of time is only of concern when the concrete performability measures are computed we define at this stage that PVs are static structures.

Definition 4.17: Performance variables for a `GS-SA net`

A performance variable p for a `GS-SA net` consists of **set of rate reward functions** ($\mathcal{R}_p$) and/or **a set of impulse reward functions** ($\mathcal{I}^p$), which are aggregated as follows:

$$\mathcal{R}_p(\vec{s}) := \sum_{r \in \mathcal{R}_p} \mathcal{R}_r(\vec{s}_{D_r}) \quad \text{and} \quad \mathcal{I}^p(\vec{s}) := \sum_{j \in \mathcal{I}^p} \mathcal{I}^j(\vec{s})$$

defined for all $\vec{s} \in \mathbb{S}_{Tan}$. Additionally we define that a PV p is of a specific **type**, so that $p.type \in \{\text{transient-state}, \text{steady-state}\}$. Thus a PV is a mapping $\mathbb{S} \times \{\text{transient-state}, \text{steady-state}\} \rightarrow (\mathbb{R}, \mathbb{R})$.

Due to the above definition $\mathcal{R}_p$ can be employed as the state-dependent rate reward function $\mathcal{R}_r(i)$ as used in Eq. 2.6 and 2.8 (p. 15). Thus the computation of $\mathcal{R}_p(t)$ and $\mathcal{R}_p(t, t + \triangle t)$ yields the user-defined (rate-oriented) performability measures of the high-level model. Analogously $\mathcal{I}^p$ can be interpreted as the sum $\sum_{j \in \mathbb{S}} \mathcal{I}^a(i,j) \cdot \lambda_{i,j}$ as employed in Eq. 2.10 and 2.13 (p. 16), so that $\mathcal{I}^p(t, t + \triangle t)$ and $\tilde{\mathcal{I}}^p$ can be computed accordingly.

Alternatively to the above definition, the aggregation of rate and impulse reward values could also be achieved, but was omitted here for simplicity.

4.2.3 Derived properties

In the following we will give some additional definitions and find some properties derived from the `GS-SA net` model and its features as specified in the previous sections. These definitions and properties will be very useful when it comes to the activity/reward-local scheme.

Language of a `GS-SA net` model

The transition system as derived from a `GS-SA net` model M directly induces a language $\mathcal{L}_{\mathcal{M}}$. This language is obviously the set of all possible transition words as generated by the transition function of M. However, a formal definition, avoiding the de-tour over the underlying *LTS*, can also be given:

Definition 4.18: Language of `GS-SA net` models

The language $\mathcal{L}_M$ as produced by a `GS-SA net` is defined as follows:

$$\mathcal{L}_M := \{\omega \in \mathcal{A}ct^* | \Delta^*(\vec{s}^{\,\epsilon}, \omega) \in \mathbb{S}\}$$

where Δ^* is the generalized state-to-state transition function as defined in Def. 4.8. Each word $\omega \in \mathcal{L}_M$ is denoted as transition word. On the level of the underlying semantic model it describes a valid sequence of state changes, starting from the initial state $\vec{s}^{\,\epsilon}$.

Concerning the notation of target states, the notation $\vec{s}^{\,\omega}$ for $\omega \in \mathcal{L}_M$ instead of the simple symbol $\vec{t}$, may be employed. This is sometimes appropriate since ω gives the sequence of activities, the execution of which led to the resp. state. Analogously we will write $\vec{s} \xrightarrow{\omega} \vec{s}^{\,\omega}$ when addressing the whole sequence of activity executions leading from states $\vec{s}$ to state $\vec{s}^{\,\omega}$. This allows us now to define the depth of a transition system.

Definition 4.19: Depth of a transition system

Let $\mathcal{L}'_M \subseteq \mathcal{L}_M$ be a subset of finite transition words, the execution sequence of which start in the initial state $\vec{s}^{\,\epsilon}$ and where each intermediate state is unique, so that

$$\forall \omega \in \mathcal{L}'_M : \left[\vec{s}^{\,\epsilon} \xrightarrow{\omega} \vec{s}^{\,\omega} \Rightarrow \left(\nexists \psi \in \mathcal{L}'_M : (\vec{s}^{\,\epsilon} \xrightarrow{\psi} \vec{s}^{\,\psi} \wedge\ |\psi| < |\omega|)\right)\right]$$

holds. The depth of the language inducing LTS is defined as $D_M := \max_{\omega \in \mathcal{L}'_M}(|\omega|)$.

Activity-local transition systems

Based on the vector-layout for states and on the set of *dependent* (*independent*) SVs we already defined the mapping of a state $\vec{s}$ to the values of the *dependent* (*independent*) SVs (cf. Eq. 4.2). Thus the *esLTS* T, as derived from the `GS-SA net` net model, can be partitioned into sets of transitions with same label l, where each state vector is reduced to the values of the activity dependent SVs by employing the above mentioned mapping. In this sense we define now the following:

Definition 4.20: Activity-local transition system

Let T be an *esLTS* as derived from a `GS-SA net` $\mathcal{M}$. For each activity $l \in \mathcal{A}ct_{\mathcal{M}}$ an activity-local transition system is defined as follows:

$$T^l := \{(\vec{x}, l, \lambda, \vec{y}) \mid\ \vec{x} := \vec{s}_{D_l} \wedge\ \vec{y} := \vec{t}_{D_l} \wedge\ (\vec{s}, l, \lambda, \vec{t}\,) \in T\}$$

Since l can either be the label of an immediate or timed activity, λ refers to a rate or weight.

The projection from T to T^l is not injective, since the abstraction from the independent SVs may result in the circumstance that an element of T^l corresponds to more than one element of T.

SV-oriented view of dependency among activities

Based on the set $\mathfrak{S}_l^{\mathcal{D}}$ one may define now a dependency relation on the set of activities:

Definition 4.21: Set of activity-dependent activities

The activities which dependent on an activity l give the set of activity-dependent activities:

$$\mathcal{A}^{D_l} := \{k \mid l, k \in \mathcal{A}ct : \mathfrak{S}_k^{\mathcal{D}} \cap \mathfrak{S}_l^{\mathcal{D}} \neq \emptyset\}.$$

In case $\mathfrak{S}_k^{\mathcal{D}} \cap \mathfrak{S}_l^{\mathcal{D}} = \emptyset$ the activities k and l are independent from each other, which can be employed, analogously to the above definition, as follows:

Definition 4.22: Set of activity-independent activities

$$\mathcal{A}^{I_l} := \{k \mid l, k \in \mathcal{A}ct : \mathfrak{S}_k^{\mathcal{D}} \cap \mathfrak{S}_l^{\mathcal{D}} = \emptyset\}.$$

where $\mathcal{A}^{D_l} = \mathcal{A}ct \setminus \mathcal{A}^{I_l}$ holds.

One may note that according to the above definition one has $l \in \mathcal{A}^{D_l}$. Each time activity l is executed, some of the *dependent* SVs for the activities of $\mathcal{A}^{D_l}$ may have changed, so that new transitions might be obtainable by subsequently executing the activities of $\mathcal{A}^{D_l}$. In contrast, the *independent* SVs are unchanged, so that for the activities $\notin \mathcal{A}^{D_l}$ no new information is generated by executing activity l. This observation is crucial and exploited under explicit SG exploration as carried out by the activity/reward-local scheme. Based on the definition of dependent activities, we can now find new definitions required in the process of explicit SG generation as well as deriving some essential properties for independent activities.

Based on the SV-oriented definition of dependency among activities, one may define the following relations:

(1) a symmetric dependency relation on $\mathcal{A}ct \times \mathcal{A}ct$:

$$(k, l) \in \mathcal{A}ct^{\mathcal{D}} \Leftrightarrow l \in \mathcal{A}^{D_k} \tag{4.5}$$

(2) a symmetric independency relation on $\mathcal{A}ct \times \mathcal{A}ct$:

$$(k, l) \in \mathcal{A}ct^{\mathcal{I}} \Leftrightarrow l \notin \mathcal{A}^{D_k} \tag{4.6}$$

Execution properties of independent activities

Concerning two activities l and k, we define the following partitioning of the set of SVs:

$$\begin{array}{ll} D_{l,k}^{DI} := \mathfrak{S}_l^{\mathcal{D}} \cap \mathfrak{S}_k^{\mathcal{I}} & D_{l,k}^{ID} := \mathfrak{S}_l^{\mathcal{I}} \cap \mathfrak{S}_k^{\mathcal{D}} \\ D_{l,k}^{DD} := \mathfrak{S}_l^{\mathcal{D}} \cap \mathfrak{S}_k^{\mathcal{D}} & D_{l,k}^{II} := \mathfrak{S}_l^{\mathcal{I}} \cap \mathfrak{S}_k^{\mathcal{I}} \end{array} \tag{4.7}$$

The pairwise intersection of the above sets is empty and their union is the set of all SVs $\mathfrak{S}$. After a suitable reordering of the state descriptor we can write $\vec{s} = (\vec{s}_{DI}, \vec{s}_{ID}, \vec{s}_{DD}, \vec{s}_{II})$. We can then distinguish the following cases concerning the execution sequences $l\,k$ and $k\,l$:

$$\begin{aligned} &\vec{s} \xrightarrow{l\,k} \vec{s}^{\,lk} \equiv \\ &\quad (\vec{s}_{DI}, \vec{s}_{ID}, \vec{s}_{DD}, \vec{s}_{II}) \xrightarrow{l} (\vec{s}^{\,l}_{DI}, \vec{s}_{ID}, \vec{s}^{\,l}_{DD}, \vec{s}_{II}) \xrightarrow{k} (\vec{s}^{\,l}_{DI}, \vec{s}^{\,k}_{ID}, \vec{s}^{\,lk}_{DD}, \vec{s}_{II}) \\ &\vec{s} \xrightarrow{k\,l} \vec{s}^{\,kl} \equiv \\ &\quad (\vec{s}_{DI}, \vec{s}_{ID}, \vec{s}_{DD}, \vec{s}_{II}) \xrightarrow{k} (\vec{s}_{DI}, \vec{s}^{\,k}_{ID}, \vec{s}^{\,k}_{DD}, \vec{s}_{II}) \xrightarrow{l} (\vec{s}^{\,l}_{DI}, \vec{s}^{\,k}_{ID}, \vec{s}^{\,kl}_{DD}, \vec{s}_{II}) \end{aligned} \tag{4.8}$$

In case $(l, k) \in \mathcal{A}ct^{\mathcal{I}} \Rightarrow D_{l,k}^{DD} = \emptyset$, yielding the following properties:

Definition 4.23: Properties of independent activities

For $(l,k) \in \mathcal{A}ct^{\mathcal{I}}$ it holds:

$$\begin{array}{ll} \text{IF } \vec{s}\,[> k \text{ THEN } \vec{s}^{\,l}\,[> k & \text{(Prop. Ia)} \\ \text{IF } \vec{s}\,[> l \text{ THEN } \vec{s}^{\,k}\,[> l & \text{(Prop. Ib)} \\ \text{IF } \vec{s}\,[> l \wedge\ \vec{s}\,[> k \text{ THEN } \vec{s}^{\,lk} = \vec{s}^{\,kl} & \text{(Prop. II)} \end{array}$$

Due to these property it is clear that the execution order of the independent activities is without significance, which is commonly also denoted as diamond property. By exploiting this one may define a well-known equivalence relation on the set of sequences of transitions, where two sequences ω and ρ are considered equivalent if and only if they can be obtained from each other by swapping adjacent independent transitions. Each equivalence class is commonly denoted as a trace [God95]. It is easy to see that one may execute independent activities on a given source state separately, the target state of the joint execution of independent activities can be obtained by simply combining the dependent sub-vectors, $\vec{s}^{\,l}_{DI}$ and $\vec{s}^{\,k}_{ID}$ in the above example. Since the diamond property holds for sequences of more than two activities which are pairwise independent, it is clear that one solely needs to generate the traces of dependent activities. The states of the traces stemming from the execution sequences of sequence-wisely independent activities can be obtained by composition. This is exploited by activity-local scheme.

Enabled dependent activities for selective exploration

Based on the definition for sets of dependent activities and based on the sets of enabled activities $\mathcal{A}_{\vec{s}\,\omega l}$ (cf. Def. 4.6), one may define the following:

Definition 4.24: Set of enabled dependent activities

The set of dependent activities enabled in a given state $\vec{s}^{\,\omega l}$ is defined as:

$$\mathcal{A}^{D_l}_{\vec{s}\,\omega l} := \mathcal{A}_{\vec{s}\,\omega l} \cap \mathcal{A}^{D_l},$$

where $\omega l \in \mathcal{L}_{\mathcal{M}}$. For the initial state we define: $\mathcal{A}^{D_\epsilon}_{\vec{s}^{\,\epsilon}} := \mathcal{A}_{\vec{s}^{\,\epsilon}}$.

During SG exploration one only executes an enabled activity in a states $\vec{s}$ once. Therefore one needs to track the states an activity was already explored in, e.g. the source states as contained within the activity-local transitions: $(\vec{s}_{D_l}, l, \lambda, \vec{t}_{D_l}) \in T^l$, where such sets may be denoted $\mathbb{E}_k$.[2] This allows one to refine the above definition:

Definition 4.25: Set of dependent activities to be executed

The set of dependent activities to be executed in a given state $\vec{s}^{\,\omega l}$ is defined as:

$$\mathcal{F}^{D_l}_{\vec{s}\,\omega l} := \{k \in \mathcal{A}^{D_k}_{\vec{s}\,\omega l} \mid \vec{s}^{\,\omega l}_{D_k} \notin \mathbb{E}_k\},$$

For the initial state we define: $\mathcal{F}^{D_\epsilon}_{\vec{s}^{\,\epsilon}} := \mathcal{A}_{\vec{s}^{\,\epsilon}}$.

This construction allows us to generate the sets of activity-local transitions T^l by following not an exhaustive - but a **selective** explicit breadth-first-search scheme. I.e. for a detected state $\vec{s}^{\,l}$, which was reached by firing action l, one generates the set of successor states by applying the activity-specific transition function δ_k for activities from $\mathcal{F}^{D_l}_{\vec{s}^{\,l}}$ only.

[2] For simplification $\mathbb{E}_k$ may record the activity-local markings of all states on which activity k was already tested in a previous step. Thus $\vec{s}^{\,l}_{D_k} \notin \mathbb{E}_k$ states that activity k was not yet tested on the activity-dependent marking of state $\vec{s}^{\,l}$.

Symbolic encodings
In the following paragraphs the main symbolic structures as employed by the activity-local scheme are discussed.

- Encodings of transitions: Each symbolic structure Z_l encoding the activity-local transition systems T^l depends only on the boolean counterparts of l's set of dependent SVs, which is defined as follows:

$$\begin{aligned} &\forall l \in \mathcal{A}ct: \\ &\quad \mathcal{V}^{\mathsf{D}_l} := \{(\mathsf{s}_1^i, \ldots, \mathsf{s}_{B_i}^i), (\mathsf{t}_1^i, \ldots, \mathsf{t}_{B_i}^i) \,|\mathfrak{s}_i \in \mathfrak{S}_l^{\mathcal{D}}\} \\ &\quad \mathcal{V}^{\mathsf{I}_l} := \{(\mathsf{s}_1^i, \ldots, \mathsf{s}_{B_i}^i), (\mathsf{t}_1^i, \ldots, \mathsf{t}_{B_i}^i) \,|\mathfrak{s}_i \in \mathfrak{S}_l^{\mathcal{I}}\} \end{aligned} \tag{4.9}$$

  In this equation, $\vec{\mathsf{s}}^{\,i}$ and $\vec{\mathsf{t}}^{\,i}$ denote those Boolean variables which encode the value of the SV $\mathfrak{s}_i$ in the source and target state of a transition $(\vec{s}, l, \lambda, \vec{t}) \in T$. For simplicity we gather now the $\mathsf{s}-$ and t-variable in different sets:
  - Set of dependent $\mathsf{s}-$variables: $\mathcal{V}_{\mathsf{s}}^{\mathsf{D}_l} := \{\mathsf{s}_j^i \in \mathcal{V}^{\mathsf{D}_l}\}$
  - Set of dependent $\mathsf{t}-$variables: $\mathcal{V}_{\mathsf{t}}^{\mathsf{D}_l} := \{\mathsf{t}_j^i \in \mathcal{V}^{\mathsf{D}_l}\}$
  - Set of independent $\mathsf{s}-$variables: $\mathcal{V}_{\mathsf{s}}^{\mathsf{I}_l} := \{\mathsf{s}_j^i \in \mathcal{V}^{\mathsf{I}_l}\}$
  - Set of independent $\mathsf{t}-$variables: $\mathcal{V}_{\mathsf{t}}^{\mathsf{I}_l} := \{\mathsf{t}_j^i \in \mathcal{V}^{\mathsf{I}_l}\}$

  The global set of boolean variables for the shared DD environment is then given by:

$$\mathcal{V}_G := \mathcal{V}_{\mathcal{A}ct} \cup \left(\bigcup_{l \in \mathcal{A}ct} \mathcal{V}_{\mathsf{s}}^{\mathsf{D}_l} \cup \mathcal{V}_{\mathsf{t}}^{\mathsf{D}_l} \right) \tag{4.10}$$

  where $\mathcal{V}_{\mathcal{A}ct}$ are the variables required for encoding the activity labels.

  For each activity its activity-local transition system T^l is represented by *Mt*-DD Z_l. Concerning the latter the following aspects should be noted:
  (1) each symbolic structure Z_l holds only transitions as induced by activity l,
  (2) each symbolic structure Z_l depends solely on the boolean variables, which encode the values of $\mathfrak{S}_l^{\mathcal{P}}$ in a source state $\vec{s}$ and the target state $\vec{t}$, so that $\mathcal{V}^{\mathsf{D}_l}$ is the set of function variables of Z_l.
  (3) for a transition $(\vec{s}, l, \lambda, \vec{t})$, the positions referring to elements of $\mathfrak{S}_l^{\mathcal{I}}$ are dropped, and
  (4) the transition weights or rates are stored within the terminal nodes of Z_l.

  In the following Z_l will therefore also be denoted as activity-local structures, which are formally defined as follows:

  Definition 4.26: Symbolic representation of an activity-local transition system

  ---

  Let a *Mt*-DD $\mathsf{Z}_l < \pi, \mathcal{V}^{\mathsf{D}_l} >$ be a symbolic representation of a pseudo-boolean function and let T^l be an activity-local transition system:

$$\mathsf{Z}_l \equiv T^l \Leftrightarrow \mathtt{Satisfy}(\mathcal{E}(l, \vec{s}_{D_l}, \vec{t}_{D_l}), \mathtt{getRoot}\mathsf{Z}_l, \mathcal{V}_{\mathsf{t}}^{\mathsf{D}_l}) = \begin{cases} \lambda & \Leftrightarrow (l, \vec{s}_{D_l}, \vec{t}_{D_l}, \lambda) \in T^l \\ 0 & \text{else} \end{cases}$$

  Function $\mathcal{E}$ supplies a binary encoding of each transition as illustrated in Sec. 3.5.1 (p. 57ff).

  Within the DD environment we assume the same ordering of the boolean variables as specified in Def. 3.14 (p. 59), where the encodings of the activity labels appear at first. However, since we have an activity-local structure for each activity, one may safely omit the activity-labels and introduce them if needed ($\mathsf{Z}_l \times \mathsf{A}_{\{\vec{\mathsf{a}}:=\mathcal{E}(l)\}}$), where the *Mt*-DD A is obtained by applying function $\mathtt{Encode}(\mathcal{E}(l), 1, \vec{\mathsf{a}})$ (cf. Algo. 3.8, p. 57).

- Encodings of tested states: The set $\mathbb{E}_l$, which is employed in Def. 4.25 for each activity, is also represented by the respective symbolic structure, denoted E_l. This structure solely depends on the set of activity l's dependent s-variables $\mathcal{V}_{\mathsf{s}}^{\mathsf{D}_l}$.

- Encodings of rate reward functions: The symbolic encodings of a rate reward's r reward function is organized in a similar fashion. Its symbolic representation solely depends on the $\mathtt{s}$-variables as contained in set $\mathcal{V}_{\mathtt{s}}^{\mathsf{D}_r}$.
- Encodings of impulse reward functions: The symbolic encoding representing the impulse reward function of an activity k ($\mathcal{I}_k^j$) is organized analogously to the encodings of transitions. I.e. the respective symbolic structure encoding an activity's impulse reward function solely depends on the variables of ($\mathcal{V}^{\mathsf{D}_l}$).

4.2.4 Boundness of models

In [Hac76] it is stated that the inclusion of more than one zero-testing arcs, which have been introduced by Agerwala [Age74] and others, gives the resulting "improved" Petri nets "Turing-power". Consequently most problems, such as boundedness and reachability are undecidable there. Since in this work we deal with arbitrary transition functions, which only need to satisfy Eq. 4.8, it is clear that all these problems are not decidable here as well. Consequently for a given `GS-SA net` model the question, if it produces a bounded *esLTS* is semi-decidable only. But if the SVs of the `GS-SA net` are known to be bounded, i.e. each SV $\mathfrak{s}_i$ can only take values from a finite set $\mathcal{W}_{\mathfrak{s}_i}$, $\mathbb{S}$ and thus T must be finite. Since for $|\mathcal{W}_{\mathfrak{s}_i}| < \infty$ trivially holds that

$$|\mathbb{S}| \leq \prod_{\mathfrak{s}_i \in \mathfrak{S}} |\mathcal{W}_{\mathfrak{s}_i}| < \infty \text{ where } \mathcal{W}_{\mathfrak{s}_i} := \{\vec{s}[i] \mid \vec{s} \in \mathbb{S}\}$$

Since $\mathfrak{s}_i$ can only take values form $\mathbb{N}$, one only needs to require therefore that there exists a maximum on $\mathcal{W}_{\mathfrak{s}_i}$ in order to enforce that $\mathbb{S}$ and T are finite sets.

Definition 4.27: k-Bounded `GS-SA net`

Let M be a `GS-SA net` model, equipped with a transition function Δ and an initial state $\vec{s}^{\,\epsilon}$:

$$M \text{ is } k\text{-bounded} \Leftrightarrow \exists k \in \mathbb{N} : k = \max_{\mathfrak{s}_i \in \mathfrak{S}} \big(\max(\mathcal{W}_{\mathfrak{s}_i})\big).$$

The maximum K_i is in general not known before hand and it is obviously semi-decidable only as well, otherwise the reachability problem would be decidable. **In the following we consider the employed high-level models to be analyzed to be bounded, but these bounds are not known a-priori to SG generation (see Sec. 3.5.1 (p. 58) for details on the handling of unknown bounds of SVs).**

4.3 The activity-local scheme: Generating symbolic representations of state graphs

Due to the existence of priorities the handling of each immediate activity in isolation is not applicable. Consequently the handling of Markovian models with immediate activities imposes additional constraints. In order to keep the discussion simple only pure Markovian high-level model specifications are considered now, i.e. we only consider models where $\mathcal{A}ct^i = \emptyset$. Issues related to immediate activities are covered in Sec. 4.6.2.

4.3.1 Main routine

The top-level algorithm of the activity-local SG generation scheme is shown in Algo. 4.1. Lines 0 - 7 contain the initialization:

(1) Z_R holds the set of states reached so far, it is initialized with the initial state $\vec{s}^{\,\epsilon}$.

Algorithm 4.1 Main routine for the activity-local SG generation scheme

```
ExploreStateGraph()
(0)   Z_R := Encode(E(s^ε), 1, V_s);
(1)   FOR k ∈ Act^m DO
(2)     Z_k := ∅;
(3)     E_k := Encode(E(s^ε_{D_k}), 1, V_s^{D_k});
(4)     IF s^ε [> k ∈ Act THEN F^{D_ε}_{s^ε} ⟵ k;
(5)   END
(6)   S-Buffer ⟵ (s^ε, F^{D_ε}_{s^ε});
(7)   T-Buffer := ∅;
(8)   DO
(9)     DO
(10)      ExploreStates();
(11)      EncodeTransitions();
(12)    END UNTIL S-Buffer = ∅
(13)    Ẑ_M := Σ_{l∈Act^m} Z_l × ⊥(G^I_l) × A_l;
(14)    Z_R := ReachabilityAnalysis();
(15)    InitNewRound(Z_R);
(16)  END UNTIL S-Buffer = ∅
(17)  Z_M := Ẑ_M · Z_R;
(18) RETURN Z_R;
```

(2) Z_k holds the activity-local transitions as induced by activity k and as generated and encoded during the explicit exploration phase. Each Z_k is initialized with the empty set.

(3) E_k encodes the sets of activity-local markings of states on which activity k was already tested for being enabled. It therefore needs solely to encode activity k's set of dependent SVs only, so that $\mathcal{V}_{\mathsf{s}}^{\mathsf{D}_k}$ is the set of function variables of E_k. Consequently E_k is initialized with the activity-dependent marking as contained within the initial state $\vec{s}^{\,\epsilon}$.

(4) $\mathcal{F}^{D_\epsilon}_{\vec{s}^{\,\epsilon}}$ is the set of activities enabled in the initial state (cf. Def. 4.25).

(5) The S-Buffer holds tuples of states and sets of activities $(\vec{s}^{\,\omega l}, \mathcal{F}^{D_l}_{\vec{s}^{\,\omega l}})$. State $\vec{s}^{\,\omega l}$ is hereby the state reached through executing the ordered activity sequence ωl. The set $\mathcal{F}^{D_l}_{\vec{s}^{\,\omega l}}$ is the set of enabled dependent activities (cf. Def. 4.25), its elements are the activities to be executed in state $\vec{s}^{\,\omega l}$

(6) The T-Buffer holds explicitly generated transitions $(\vec{s}^{\,l}, k, \lambda, \vec{s}^{\,lk})$ to be encoded and inserted into the respective symbolic activity-local structure Z_k.

In the inner DO-UNTIL loop (lines 9 - 12) procedures `ExploreStates` and `EncodeTransitions` are called in an alternating fashion in order to carry out explicit SG exploration and the encoding of the detected transitions. If a fixed point is reached, i.e. all sequences of dependent activities starting from the initial state(s) are extracted and no new states for further exploration are detectable, symbolic composition takes place (line 13). Since the obtained symbolic structure $\widehat{\mathsf{Z}_M}$ encodes a set of potential transitions, it is necessary to perform symbolic reachability analysis (line 14). Symbolic composition and reachability analysis gives one now all states reached so far, but where some of them are generated only on the symbolic level. Such composed states may trigger new model behavior, consequently re-initialization must take place. This is realized by calling routine `InitNewRound`, (line 15), which searches for such states and inserts them together with the respective activity into the S-Buffer. If states, triggering new model behavior exist, a new round of explicit SG exploration and encoding follows, i.e. one re-enters the outer DO-UNTIL loop (lines 8 - 16). In case such states do not exist anymore, the activity-local scheme is finished and all reachable transitions of a high-level model's underlying *sLTS* have been computed (lines 17).

Algorithm 4.2 Procedures for explicit SG generation and encoding

ExploreStates()
(0) WHILE $\mathbb{S}$-Buffer $\neq$ *empty* DO
(1) $(\vec{s}^{\,l}, \mathcal{F}^{D_l}_{\vec{s}^{\,l}}) \longleftarrow$ $\mathbb{S}$-Buffer;
(2) FOR $k \in \mathcal{F}^{D_l}_{\vec{s}^{\,l}}$ DO
(3) $\vec{s}^{\,lk} := \delta_k(\vec{s}^{\,l})$;
(4) $\lambda := \Lambda_k(\vec{s}^{\,l}_{D_k}, k, \vec{s}^{\,lk}_{D_k})$;
(5) $\mathbb{T}$-Buffer$\longleftarrow (\vec{s}^{\,l}, k, \lambda, \vec{s}^{\,lk})$;
(6) END
(7) END
(8) RETURN ;

(A) Routine carrying out explicit SG exploration for a dedicated set of activities

EncodeTransitions()
(0) WHILE $\mathbb{T}$-Buffer $\neq$ *empty* DO;
(1) $(\vec{s}, l, \lambda, \vec{s}^{\,l}) \longleftarrow$ $\mathbb{T}$-Buffer;
(2) $\mathcal{F}^{D_l}_{\vec{s}^{\,l}} := \emptyset$;
(3) FOR $k \in \mathcal{A}^{D_l}$ DO;
(4) IF $\vec{s}^{\,l}_{D_k} \notin \mathsf{E}_k \wedge \vec{s}^{\,l}$ [$> k$
(5) THEN $\mathcal{F}^{D_l}_{\vec{s}^{\,l}} := \mathcal{F}^{D_l}_{\vec{s}^{\,l}} \cup \{k\}$;
(6) $\mathsf{E}_k := \mathsf{E}_k + \texttt{Encode}(\mathcal{E}(\vec{s}^{\,l}_{D_k}), 1, \mathcal{V}^{\mathsf{D}_k}_{\mathsf{s}})$;
(7) END
(8) IF $\mathcal{F}^{D_l}_{\vec{s}^{\,l}} \neq \emptyset$
(9) THEN $\mathbb{S}$-Buffer $\longleftarrow (\vec{s}^{\,l}, \mathcal{F}^{D_l}_{\vec{s}^{\,l}})$;
(10) $\mathsf{Z}_l := \mathsf{Z}_l + \texttt{Encode}(\mathcal{E}(\vec{s}_{D_l}, \vec{s}^{\,l}_{D_l}), \lambda, \mathcal{V}^{\mathsf{D}_l})$;
(11)END
(12) RETURN ;

(B) Routine carrying out symbolic encoding and preparing next round of explicit exploration

4.3.2 Explicit state graph generation and encoding

For generating the sets of activity-local transitions T^l we follow a *selective* breadth-first-search strategy, i.e. for a detected state $\vec{s}^{\,l}$, which was reached by firing action l, we generate the set of successor states by applying the activity-specific transition function δ_k for each activity $k \in \mathcal{F}^{D_l}_{\vec{s}^{\,l}}$, where $\mathcal{F}^{D_l}_{\vec{s}^{\,l}}$ is the set of dependent enabled activities to be fired (cf. Def. 4.25). The set $\mathbb{E}_k$ as employed in Def. 4.25 is hereby represented by *Mt*-DD E_k. It encodes those activity-local markings on which activity k was already tested (successfully or not). One could also test if $\vec{s}^{\,l}_{D_k} \in \mathsf{Z}_k$, as source or as target state: $\vec{s}^{\,l}_{D_k} \in (p\texttt{ZAbstract}(\mathsf{Z}_k, \vec{\mathsf{t}}, \vee)\{\vec{\mathsf{s}} \leftarrow \vec{\mathsf{t}}\} \vee p\texttt{ZAbstract}(\mathsf{Z}_k, \vec{\mathsf{s}}, \vee))$. In such a case, one may repeatedly test states where $\vec{s}[\not> k$, but doing so would impose a minor run-time overhead. Consequently E_k is initialized with the model's initial state $\vec{s}^{\,\epsilon}$ (cf. line 3 of Algo. 4.1). The explicit SG generation and encoding itself is realized with the help of two complementary procedures `ExploreStates` and `EncodeTransitions`. Their pseudo-code is specified as Algo. 4.2.A and B.

Explicit SG exploration

The task of routine `ExploreStates` (Algo. 4.2.A), is to execute all activities as contained in set $\mathcal{F}^{D_l}_{\vec{s}}$ on a state $\vec{s}^{\,l}$ and to insert the established transitions into the $\mathbb{T}$-Buffer. To do so it reads tuples, consisting of a state and a list of activities from the $\mathbb{S}$-Buffer (line 1). For each activity $k \in \mathcal{F}^{D_l}_{\vec{s}}$ the successor state $\vec{s}^{\,lk}$ and the corresponding rate λ are computed (lines 3 and 4). The obtained transition tuple $(\vec{s}^{\,l}, k, \lambda, \vec{s}^{\,lk})$ is then inserted into the $\mathbb{T}$-Buffer (line 5). Once $\mathbb{S}$-Buffer is empty, routine `ExploreStates` terminates.

Encoding the SG

At first the complementary routine `EncodeTransitions` (Algo. 4.2.B) reads a transition from the $\mathbb{T}$-Buffer (line 1), secondly it generates the set $\mathcal{F}^{D_l}_{\vec{s}^{\,l}}$ (FOR-loop of lines 3 - 7). In line 6 the activity-local marking of state $\vec{s}^{\,l}$ with respect to k is inserted into the symbolic structure E_k. Hereby functions $\mathcal{E}$ and `Encode` are the functions as already employed in Sec. 3.5, where $\mathcal{E}$ converts a state into a bit-string and `Encode` generates the respective symbolic representation. As third step `EncodeTransitions` inserts the tuples of state/activity-lists into the $\mathbb{S}$-Buffer (line 9) as to be employed in the next round of explicit exploration. As final step the transition itself is encoded and inserted into the symbolic structure representing the respective activity-local transition system (line 10). `EncodeTransitions` terminates as soon as all transitions are processed and $\mathbb{T}$-Buffer is empty.

By executing procedures `ExploreStates` and `EncodeTransitions` in an alternating sequential fashion, the scheme will reach a point where `EncodeTransitions` has been executed and the $\mathbb{S}$-Buffer is still empty. This means that the algorithm has visited all states reachable from the initial state(s) through sequences of dependent activities. However, so far we have not considered the combined execution of independent activities. The shuffled execution of independent activities is realized on the level of the symbolic representation.

4.3.3 Symbolic state graph composition

At the end of explicit SG exploration and encoding, when the procedures `ExploreStates` and `EncodeTransitions` have reached a local fixed point, $n_{\mathcal{A}ct}$ symbolic structures Z_l are generated. Each of these symbolic structures depends solely on those boolean variables encoding the activity-dependent SVs $\mathfrak{S}_l^{\mathcal{D}}$ **before** and **after** the respective activity's execution (cf. Eq. 4.9, p. 851). Before composition can take place, each of the symbolic structures Z_l needs to be supplemented by the set of Boolean variables $\mathcal{V}^{\mathsf{I}_l}$ encoding the set of independent SVs, yielding the set of *potential* transitions as induced by activity l. Since a SV $s_i \in \mathfrak{S}_l^{\mathcal{I}}$ does not change its value when activity l is executed, i.e. it remains stable, one may employ the identity structures for supplementation. Since on the level of Boolean variables $\vec{\mathsf{s}}^{\,i}, \vec{\mathsf{t}}^{\,i} \in \mathcal{V}^{\mathsf{I}_l}$, encode the activity-independent SVs $\mathfrak{S}_l^{\mathcal{I}}$ in the source and target state (cf. Eq. 4.9), the symbolic structure $\mathsf{1}(\mathcal{V}^{\mathsf{I}_l})$, or $\mathsf{1}_l$ for short, (cf. Def 3.14, p. 61), will deliver the required symbolic representation. After the activity-local transition systems have been supplemented, they can be combined in order to obtain a symbolic representation of the overall transition system T:

$$\mathsf{Z}^p := \sum_{l \in \mathcal{A}ct} \mathsf{Z}_l \times \mathsf{1}_l \times \mathsf{A}_l \tag{4.11}$$

Hereby A_l represents the binary encoded activity label l.[3] Due to the use of the identity structure over the values ranging from 0 to $2^{\lceil \log_2(K_i) \rceil} - 1$ for each boolean s- and t-vector i, the composed transition system Z^p constructed may not contain reachable transitions only. Consequently at this point it is necessary to perform symbolic reachability analysis in order to restrict Z^p to the set of reachable transitions.

4.3.4 Symbolic reachability analysis

In the following a new variant for carrying out symbolic reachability analysis will be presented. However, at first the standard approach as published in previous works [PRCB94, Sie01] and as employed in tools such as Caspa [KSW04] and Prism [Pri] will be discussed. In both variants one obtains the set of reachable states, which allows one to restrict the set of potential transitions to the actual reachable ones.

Standard breadth-first-search symbolic reachability analysis

The algorithm for standard breadth-first-search (bfs) symbolic reachability analysis itself employs mainly the following four symbolic structures:

(1) Z_U representing the set of *unexplored states*,

(2) Z^p representing the set of *potential* transitions as induced by the $n_{\mathcal{A}ct}$ activities. For simplicity we will ignore the activity-labels and the transition rates, and assume that Z^p only encode the potential transition relation $T \subseteq \mathbb{S} \times \mathbb{S}$.[4]

(3) Z_R, representing the set of *reached states*, (initialized in line 0 of Algo 4.1, p. 87) and

[3] Since the activity-local transition systems are stored individually, these labels can also be omitted and introduced when needed, e.g. when computing impulse rewards.

[4] As pointed out in chapter 3 this is justified, since our symbolic algorithms are also capable to employ binary operators to pseudo-boolean functions (cf. Sec. 3.4.2, p. 50ff).

Algorithm 4.3 Variants of symbolic reachability analysis

ReachabilityAnalysis()
(0) $\mathsf{Z}_U := \mathsf{Z}_R$;
(1) $\mathsf{Z}^p := \sum_{l \in \mathcal{A}ct} \mathsf{Z}_l \times \mathbb{1}_l$;
(2) DO
(3) $\quad \mathsf{Z}_{tmp} := \mathsf{Z}^p \cdot \mathsf{Z}_U$;
(4) $\quad \mathsf{Z}_{tmp} := p\texttt{ZAbstract}(\mathsf{Z}_{tmp}, \mathcal{V}_\mathsf{s}, +)$;
(5) $\quad \mathsf{Z}_{tmp} := \mathsf{Z}_{tmp} \setminus \mathsf{Z}_R$;
(6) $\quad \mathsf{Z}_R := \mathsf{Z}_R + \mathsf{Z}_{tmp}\{\mathcal{V}_\mathsf{s} \leftarrow \mathcal{V}_\mathsf{t}\}$;
(7) $\quad \mathsf{Z}_U := \mathsf{Z}_{tmp}\{\mathcal{V}_\mathsf{s} \leftarrow \mathcal{V}_\mathsf{t}\}$;
(8) END UNTIL $\mathsf{Z}_U = \emptyset$
(9) RETURN Z_R;

(A) Breadth-first-search symbolic reachability analysis as proposed in [PRCB94, Sie01]

ReachabilityAnalysis()
(0) $\mathsf{Z}_U := \mathsf{Z}_R$;
(1) FOR $k \in \mathcal{A}ct$ DO
(2) $\quad \widetilde{\mathsf{Z}_k} := \mathsf{Z}_k \times \mathbb{1}_k$;
(3) END
(4) DO
(5) $\quad \mathsf{Z}_R := \mathsf{Z}_R + \mathsf{Z}_U$;
(6) $\quad$ FOR $k \in \mathcal{A}ct$ DO
(7) $\quad\quad \mathsf{Z}_{tmp} := p\texttt{ZAbstract}(\widetilde{\mathsf{Z}_k} \cdot \mathsf{Z}_U, \mathcal{V}_\mathsf{s}, +)$;
(8) $\quad\quad \mathsf{Z}_{tmp} := \mathsf{Z}_{tmp} \setminus \mathsf{Z}_R$;
(9) $\quad\quad \mathsf{Z}_U := \mathsf{Z}_U + \mathsf{Z}_{tmp}\{\mathcal{V}_\mathsf{s} \leftarrow \mathcal{V}_\mathsf{t}\}$;
(10) $\quad$ END
(11) $\quad \mathsf{Z}_U := \mathsf{Z}_U \setminus \mathsf{Z}_R$;
(12) END UNTIL $\mathsf{Z}_U = \emptyset$
(13) RETURN Z_R;

(B) Quasi depth-first-search symbolic reachability analysis

(4) Z_{tmp} representing the set of states detected in the current iteration.

In line 0 the initialization of Z_U with the states already reached so far is done. In line 1 of Algo. 4.3.A one executes the composition scheme (cf. line 13 of Algo. 4.1). This gives one the symbolic structure Z^p representing the set of *potential* transitions. A set-oriented bfs-scheme is then realized by the DO-UNTIL loop of lines 2 - 8, since the multiplication of Z_U (unexplored states) and Z^p (potential transitions) delivers all transitions emanating from the states contained in the symbolic structure Z_U. The subsequent re-movement of the source states by executing an existential abstraction on the variables $\vec{\mathsf{s}}$ (line 4) gives one finally the set of states reachable from the states of Z_U in one step, commonly denoted as image. Before one enters now the next iteration, the set of reachable states and the set of unexplored states as represented by Z_R and Z_U are updated (line 6 and 7). Once the set of unexplored states is empty, the set of all reachable states has been computed.

Quasi-dfs symbolic reachability analysis

The step of image computation as carried out in Algo. 4.3.A can be understood as being set-oriented and parallel. Since Z_U may represent more than one state, and one obtains all successor states in one step, one operation respectively ($\mathsf{Z}_U \cdot \mathsf{Z}^p$, as executed in line 3). However, the required operations are all carried out by the `Apply`-algorithms or variants thereof. Thus, the larger and denser the symbolic structures are the more hanging recursive calls to `Apply` there are and the more `op`-cache replacements may occur. Therefore we propose the following improvements:

(1) Replace the "*parallel*" set-oriented scheme (line 3 of Algo. 4.3.A) by a *sequential* set-oriented scheme (lines 6 - 10 of Algo. 4.3.B).

(2) Update the set of unexplored states as soon as possible (line 9 of Algo. 4.3.B).

Algorithm 4.3.B realizes an activity-wise (sequential) application of the potential transition functions in combination with an early update of Z_U. This gives one a set-oriented mixture of a bfs - and a depth-first-search (dfs) scheme. We denote this scheme as quasi-dfs scheme, since activities are executed on a *set of* states sequentially, rather than all at once as before.

Doing so leads to a significant reduction of the number of iterations of the main (outer) DO-UNTIL loop if compared with the original approach of Algo. 4.3.A. However, it is worth noting that in case the symbolic structure Z_U would not be updated with the newly reached states directly (line 9 of Algo. 4.3.B), but outside the inner FOR-loop, one would obtain exactly

Algorithm 4.4 Re-initialization of explicit SG exploration and encoding

```
InitNewRound(Z_R)
(0)  FOR k ∈ Act DO
(1)    Z_tmp := Z_R \ E_k;
(2)    WHILE Z_tmp ≠ ∅ DO
(3)      Z_s ←s⃗— Z_tmp;
(4)      s⃗ := E^-1(Encode^-1(Z_s));
(5)      IF s⃗ [> k THEN S-Buffer ⟵ (s⃗, {k})
(6)      Z_tmp := Z_tmp \ pZAbstract(Z_s, V_s^{l_k}, +);
(7)      E_k := E_k + Encode(E(s⃗_{D_l}), 1, V_s^{D_l});
(8)    END
(9)  END
(10) RETURN ;
```

the same number of iterations of the main (outer) DO-UNTIL loop for both reachability algorithms, but with lower run-times in case of the new algorithm.

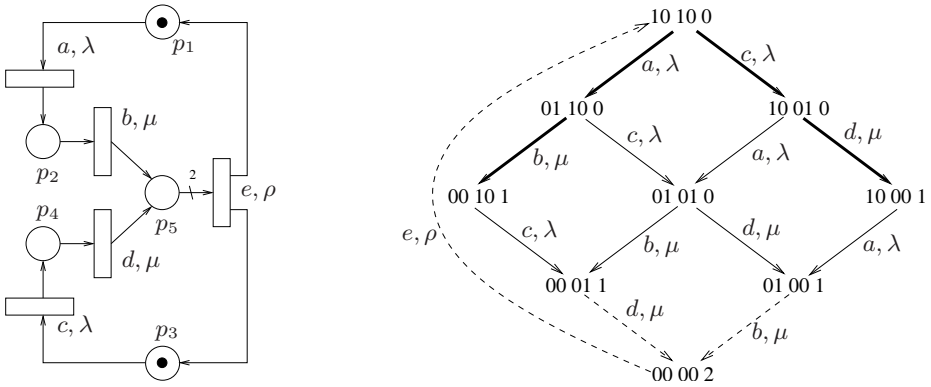
In the spirit of a greedy-heuristic, the early update-strategy can be taken a step further. By nesting line 7-9 in an additional loop, one is enabled to execute the current activity until a local fixed point is reached, i.e. until no new state is visited. This strategy will decrease the number of iterations of the main (outer) DO-UNTIL loop further. However, experience shows that the additional gained speed-up is strongly model-dependent and sometimes relatively low.

4.3.5 Re-initialization of the scheme

After symbolic composition and symbolic reachability analysis have taken place, one ends up with all states, reachable from the initial state so far. However, the scheme only explicitly generated sequences of dependent activities, where symbolic composition gives their shuffled execution, but on the level of the symbolic representation. On the level of the high-level model the states resulting from the joint execution of independent activities have not been considered yet. As a consequence, activity sequences requiring independent activity sequences as prefix may not be yet detected. Therefore algorithm `InitNewRound`, as specified as Algo. 4.4 checks for each activity individually, if there is a composed state triggering new model behavior. In case it detects such a state, it inserts it together with the respective activity into the $\mathbb{S}$-Buffer. To do so, `InitNewRound` takes the set of reachable states as input parameter (Z_R). In lines 1 - 8 the algorithm determines those reachable states on which a given activity has not yet been tested. For each of these states it tests now, if the respective activity is enabled or not, since in case it is, new model behavior will be obtained. To do so, pairs of such states and enabled activities are inserted into the $\mathbb{S}$-Buffer (line 5), yielding the input for the next round of explicit SG exploration, encoding, and symbolic composition and reachability analysis. Considering only positions of dependent SVs imposes an equivalence class on the set of states. Therefore routine `InitNewRound` only considers states differing on the positions of dependent SVs (line 6) reducing the number of states to be tested significantly, since one only needs to take care of one state per equivalence class. If the $\mathbb{S}$-Buffer is still empty after the execution of `InitNewRound`, the activity-local scheme has reached a global fixed point and a symbolic representation of the complete SG has been generated.

4.3.6 Example

Fig. 4.1 shows a simple SPN and its underlying *sLTS*. The activity-local structures as obtained under the interleaved execution of `ExploreStates` and `EncodeTransitions` are shown in Fig. 4.2.A. The binary encodings of the transition as produced by the activity-local scheme are given in Fig. 4.2.B. The symbolic representation of the set of reachable states, as well as of the *sLTS*, as obtained after the final execution of symbolic composition and symbolic reachability analysis is shown in Fig. 4.3. The *Mt*-DDs in the illustrations are



(A) A stochastic Petri net (B) The corresponding *sLTS*

Figure 4.1: A SPN and its underlying *sLTS*

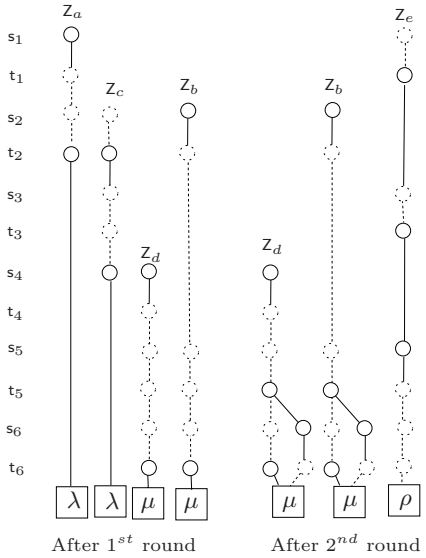
ordered, i.e. on all paths from the root to a terminal node we have the same variable ordering, and they are reduced, i.e. all isomorphic substructures have been merged. In the ADDs, a dashed (solid) arrow indicates the value assignment 0 (1) to the corresponding Boolean variable on the respective path. The nodes printed in dashed lines are those which get eliminated when applying the *zero-suppressing* reduction rule for ZDDs. Hereby the variables of all other skipped levels, i.e. the levels where absolutely no nodes appear, refer to variables which are not function-variables of the respective ZDD.

In the following the activity-local scheme illustrated so far will be recapitulated by means of this example, where one may ignore the rate information for the moment, since it is irrelevant for the following discussion.

Starting from the initial marking (10100) of the SPN of Fig. 4.1.A, the activity-local scheme will explore those transitions explicitly which are drawn by fat arrows in the *sLTS* of Fig. 4.1.B. As an example, transition 10100 $\xrightarrow{a}$ 01100 will be explored and then encoded in the activity-local symbolic structure Z_a of activity a as 10**** $\longrightarrow$ 01**** (cf. Fig. 4.2.A), where the symbol * denotes a skipped position, since the respective boolean variables are not function-variables of Z_a (cf. Fig. 4.1.A: only p_1 and p_2 belong to the set of dependent SVs of activity a). The transitions drawn in Fig. 4.1.B by regular arrows are the ones which are generated during the composition of the activity-local symbolic structures, which can be seen as a cross product construction followed by symbolic reachability analysis as realized by one of the algorithms presented as Algo. 4.3 and as called before procedure `InitNewRound` is executed.

We will now explain why the transitions drawn as dashed arrows in the figure are not explicitly generated during the first round of exploration, i.e. the reason why more than one round of explicit exploration is required: Consider, for example, transitions caused by activity d: In the first round the algorithm explicitly generates the transition 10010 $\xrightarrow{d}$ 10001, which is encoded in the activity-local symbolic structure Z_d as ***100 $\longrightarrow$ ***001 (cf. Fig. 4.2). The cross product construction yields any transition +++100 $\xrightarrow{d}$ +++001, where the +-positions are arbitrary *but stable*. But the composition does not yield the transition 000101 $\rightarrow$ 00010, which refer to the dashed transition 00011 $\xrightarrow{d}$ 00002 in the *sLTS*. During procedure `InitNewRound`, one detects that state 00011 is reachable and that activity d has not yet been tested in states of the type ***11. Therefore the tuple (00011,d) will be inserted into the $\mathbb{S}$-Buffer at this point, and this dashed transition (as well as the other two dashed transitions) will be explored in the second round. After this second round `InitNewRound` will not insert any (state/activity)-tuples into the $\mathbb{S}$-Buffer, and the scheme reaches a global fixed point and thus terminates.

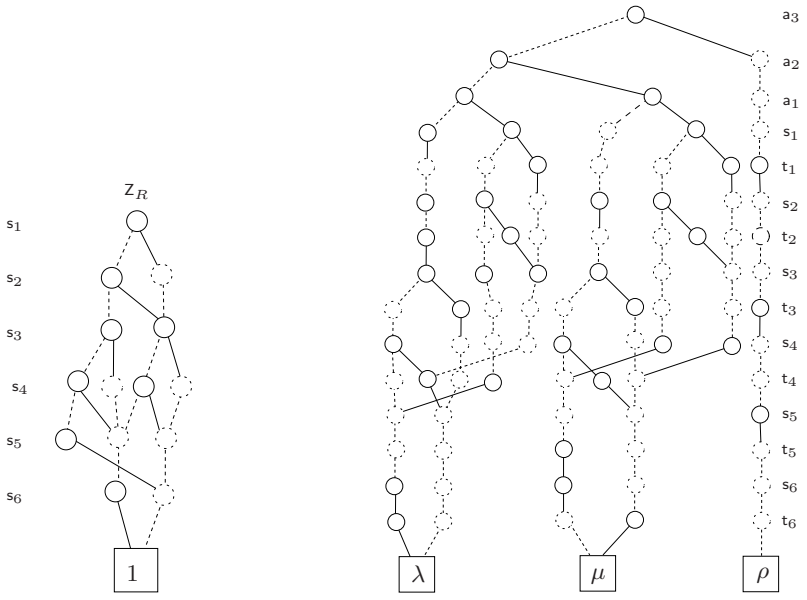
(A) *p*ZDDs representing the act.-local SGs[5]



(B) Binary encodings

l	$\vec{a}$	$\vec{s}$	$\vec{t}$	f_M	Round
	$a_1a_2a_3$	$s_1s_2s_3s_4s_5s_6$	$t_1t_2t_3t_4t_5t_6$		
a	000	10****	01****	λ	1
c	001	**10**	**01**	λ	1
b	010	*1**00	*0**01	μ	1
		*1**01	*0**10		2
d	011	***100	***001	μ	1
		***101	***010		2
e	100	0*0*10	1*1*00	ρ	2

Figure 4.2: Activity-local structures and binary encodings



(A) Reachable states (B) SLTS

Figure 4.3: Symbolic representation of the set of reachable states and the *sLTS*

[5] For illustration purpose nodes belonging to *0-sup.* variables are also given, the dashed ones. Consequently only levels referring to non-function variables are skipped within the *p*ZDDs.

4.4 Completeness and correctness of the scheme

In this section the correctness and completeness of the activity-local scheme will be shown. This is achieved in three steps:

- The correctness of the symbolically represented activity-local transition systems is due to the fact that the value changes on positions referring to SVs of $\mathfrak{S}_l^{\mathcal{P}}$ are directly derived from the high-level model's execution (line 3 and 4 of Algo. 4.2, p. 88). This allows to concentrate on the completeness of the explicit generation and encoding scheme, where Theorem 4.4 (p. 96) states that each of the obtained symbolic representations (Z_l^D) of an activity-local transition system is complete (and according to the above argument also correct).

- Secondly we show that the symbolic composition scheme, as developed in Eq. 4.11, delivers a superset of the transitions of a high-level model's underlying *sLTS*, which will find its formalization in Theorem 4.5 (p. 96).

- As third step we will prove Theorem 4.7 (p. 98), which states that the symbolic reachability analysis delivers the correct set of reachable states (Z_R), even though it employs the superset of transitions.

Based on the correctness and completeness of the activity-local scheme, of the symbolic composition scheme and of the symbolic reachability analysis we can conclude, that for a high-level model's underlying *sLTS* T, represented by a symbolic structure $\mathsf{Z}^T{<}\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}}{>}$ the following must hold:

$$T \equiv \mathsf{Z}^T{<}\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}}{>} = \left(\sum_{l \in \mathcal{A}ct} \mathsf{A}_l \times \mathsf{Z}_l^D \times \mathbb{1}_l \right) \cdot \mathsf{Z}_R,$$

where the symbolic structures on the right hand side of the above equation are the ones as delivered by the activity-local scheme, namely the set of potential transitions constructed by the expression in parenthesis and the set of reachable states (Z_R). The restriction of the potential transition system to the real one by multiplying the respective symbolic structures is possible, since the symbolic composition scheme solely constructs un-reachable and reachable transitions, but no false positives (cf. Sec. 4.4.3, p. 98ff). Thus the above equation immediately implies, that at termination the activity-local scheme constructed a valid symbolic representation of a high-level model's underlying *sLTS*.

Weights and rates of transitions are generated by executing the high-level model (line 4 of Algo. 4.2, p. 88), where its is required that the activity-local weight or rate-returning function only takes parameters from $\mathfrak{S}_l^{\mathcal{P}}$ (cf. Def. 4.11 and 4.12). Furthermore weights and rates are stored within the terminal nodes, where the composition scheme employs the identity function on the variables encoding SVs of $\mathfrak{S}_l^{\mathcal{T}}$ (cf. Eq. 4.11). Thus, weights and rates of transitions are not changed when the potential transition system is generated. Consequently one only needs to deal in the following with the case of an *LTS* and ignore weight - and rate information.

4.4.1 Generation scheme

The explicit exploration undertaken may be partial. Therefore it needs to be shown that our approach delivers the complete activity-local *sLTS* T^l for each activity as defined in Def. 4.20 (p. 82). A single execution of the DO-UNTIL loop of line 8-16 of Algo. 4.1 (p. 87) will be denoted in the following as a round. Since the set of partial state markings E_k, the set of activity-local transitions Z_k, as well as the set of reachable states Z_R depend on the number of rounds executed, they are now indexed accordingly. I.e. E_k^i, Z_k^i and Z_R^i are the respective symbolic structures as obtained after the i'th round has been executed. At the beginning Z_R^0 contains the initial state $\vec{s}^{\,\epsilon}$, the set E_k^0 its activity-local marking $\vec{s}_{D_k}^{\,\epsilon}$, whereas Z_k^0 is the empty set (line 0-5 of Algo. 4.1, p. 87). Since we deal with finite SGs, the scheme

will terminate after K rounds have been executed.

Lemma 4.1: *For some state $\vec{s}$ the activity-local scheme will generate all sequences of pairwise dependent activities.*

Proof: Let within the j'th round exploration and encoding be executed n_j-times (line 9-12 of Algo. 4.1), yielding the following sequences of activity firings:

$$\vec{s} \xrightarrow{\alpha_1} \vec{s}^{\,\alpha_1} \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n_j}} \vec{s}^{\,\alpha_1,\ldots\alpha_{n_j}}$$

where $(\alpha_{i-1}, \alpha_i) \in \mathcal{A}ct^{\mathcal{D}}$ and $\vec{s}^{\,\alpha_1\ldots\alpha_{i-1}}$ $[> \alpha_i$ for $0 < i \leq n_j$ holds. For simplification, let $k := \alpha_i, l := \alpha_{i+1}$ and $\omega := \alpha_1 \ldots \alpha_{i-1}k$. The following two cases appear when state $\vec{s}^{\,\omega}$ is visited:

$$\begin{array}{ll} \text{(A) } l \in \mathcal{F}^{D_k}_{\vec{s}^{\,\omega}} \Rightarrow \vec{s}^{\,\omega}_{D_l} \xrightarrow{l} \vec{s}^{\,\omega l}_{D_l} \in \mathsf{Z}^j_l & \text{(transition will be explored)} \\ \text{(B) } l \notin \mathcal{F}^{D_k}_{\vec{s}^{\,\omega}} \Rightarrow \vec{s}^{\,\omega}_{D_l} \xrightarrow{l} \vec{s}^{\,\omega l}_{D_l} \in \mathsf{Z}^{<j}_l \vee \nexists \vec{s}^{\,\omega}_{D_l} \xrightarrow{l} \vec{s}^{\,\omega l}_{D_l} & \text{(transition already explored or does not exist)} \end{array} \tag{4.12}$$

The above equation simply states that if there is an execution sequence of pairwise dependent activities ($(l,k) \in \mathcal{A}ct^{\mathcal{D}}$) starting from state $\vec{s}$, it will be explored and this only once (cf. DO-UNTIL loop of line 9-12 of Algo. 4.1 in combination with Algo. 4.2.A and line 4-6 of Algo. 4.2.B. (p. 88)).

The above lemma holds for all sequences as long as the pair $(\vec{s}, \alpha_1)$ is entered into the $\mathbb{S}$-Buffer.

Lemma 4.2: *The re-initiation of the activity-local scheme will trigger the exploration of all states contained in Z^j_R, as long as their activity-local marking have not been exposed to the respective activity yielding (new) model behavior, which is not already contained in Z^j_T.*

Proof: Let us assume now that within round j explicit exploration and encoding has reached its local fixed point ($\nexists \alpha_{n_j} \in \mathcal{A}ct : \mathcal{F}^{D_{\alpha_{n_j}}}_{\vec{s}^{\,\alpha_1\cdots\alpha_{n_j}}} \neq \emptyset$). Now symbolic composition takes place, yielding the joint execution sequences of all activity sequences explicitly extracted so far and encode in Z^j_T (cf. Sec. 4.2.3: Execution properties of independent activities). The subsequent execution of a symbolic reachability analysis gives then the current set of reachable states Z^j_R. –Symbolic reachability is necessary since as it will be discussed below, symbolic composition yields the potential transition system.– The next step is re-initialization, where the states of Z^j_R serve as input and where for composed states triggering new model behavior, the following holds:

$$(\exists \vec{s} \in \mathsf{Z}^j_R : \vec{s} \xrightarrow{l} \vec{t} \in T \wedge\ \vec{s}_{D_l} \xrightarrow{l} \vec{t}_{D_l} \notin \mathsf{Z}^j_l) \Rightarrow \vec{s}_{D_l} \notin \mathsf{E}^{j-1}_l \wedge\ \vec{s}\,[> l \tag{4.13}$$

The execution of routine `InitNewRound` (Algo. 4.4, p. 91) yields then:

$$\mathsf{E}^j_l := \mathsf{E}^{j-1}_l + \texttt{Encode}(\mathcal{E}(\vec{s}_{D_l}), 1, \mathcal{V}^{\mathsf{D}_l}_{\mathsf{s}}) \text{ and } l \rightarrow \mathcal{F}^{D_\epsilon}_{\vec{s}} \tag{4.14}$$

where E^{j-1}_l is the set of activity-local markings tested explicitly for enabling activity l, before the inner WHILE-loop of `InitNewRound` is executed (line 2-7 Algo. 4.4). $\mathcal{F}^{D_\epsilon}_{\vec{s}}$ is a singleton, containing an activity to be explicitly explored in state $\vec{s}$, where this state/activity pair is inserted into the $\mathbb{S}$-Buffer (line 5) and to be processed by routine `ExploreStates` (Algo. 4.2.) in the next round $j+1$, so that

$$\vec{s}_{D_l} \xrightarrow{l} \vec{t}_{D_l} \in \mathsf{Z}^{j+1}_l \tag{4.15}$$

holds. I.e. according to Lemma 4.1 and 4.2 symbolic composition, symbolic reachability and re-initialzation trigger another round $(j+1)$, which starts once again with the n_{j+1}-fold explicit exploration and encoding of activity-execution sequences, where $(\alpha_{i-1}, \alpha_i) \in \mathcal{A}ct^{\mathcal{D}}$ holds.

Lemma 4.3: *At termination the activity-local scheme explored and generated activity-sequences up to the length of the depth (cf. Def. 4.19) of the high-level model's underlying* LTS*.*

Proof: According to Lemma 4.1 the activity-local scheme explicitly explores all execution sequences of pairwise dependent activities (ω) starting in a state $\vec{s}$, so that at the end of round j s_j sequences, with the max. length n_j have been extracted. Symbolic composition and symbolic reachability will then generate sequences of max. length $N_j := \sum_{i:=1}^{j} n_i$ (see discussion of diamond property in Sec. 4.2.3: Execution properties of independent activities). Since this procedure is repeated until $\vec{s}^{\omega}$ does not trigger any new model behavior, an activity execution sequence with the maximal length N_K is generated. However, this must be the depth of the *LTS*, since otherwise, according to Lemma 4.2, $\vec{s}^{\rho}$ with $|\rho| = N_K$ would trigger new model behavior.

For wrapping up the above discussion, let us assume that the activity-local scheme terminates after K rounds and $\exists \vec{s}^{\omega} \xrightarrow{l} \vec{s}^{\omega l} \in T$ and $\vec{s}^{\omega}_{D_l} \xrightarrow{l} \vec{s}^{\omega l}_{D_l} \notin \mathsf{Z}_l^K$. According to Lemma 4.3 the activity-local scheme generates execution sequences of the length $N_K := \sum_{i:=1}^{K} n_i$ Now the following cases have to be considered:

(1) $|\omega l| < N_K$: Lemma 4.1 and 4.2 yield that state $\vec{s}^{\omega}$ is explored in round $j < K$. Thus $\vec{s}^{\omega}_{D_l} \xrightarrow{l} \vec{s}^{\omega l}_{D_l} \in \mathsf{Z}_l^j$ and $\mathsf{Z}_l^j \subseteq \mathsf{Z}_l^K$ gives $\vec{s}_{D_l} \xrightarrow{l} \vec{s}^{\omega l}_{D_l} \in \mathsf{Z}_l^K$ (contradiction !).

(2) $|\omega l| > N_K$: Obviously N_K is not the depth of T, but according to Lemma 4.3 the activity-local scheme stops only if all sequences up to the depth of the *sLTS* have been generated. Consequently this case can never appear.

From this discussion, we can conclude:

Theorem 4.4: *The selective bfs scheme for explicitly exploring states (combined with the symbolic composition scheme) as employed by the activity-local scheme is complete, i.e.:*

$$\nexists \vec{s} \xrightarrow{l} \vec{t} \in T : \vec{s}_{D_l} \xrightarrow{l} \vec{t}_{D_l} \notin \mathsf{Z}_l^K$$

It is essential to note that the above line of argumentation assumed that symbolic composition and reachability analysis delivers the correct and complete set of states as reachable form the initial state $\vec{s}^{\epsilon}$ and as generated up to round j. This completeness and correctness will be discussed next. However, the fact that the generation of Z_R and Z_T is possibly divided over several rounds is irrelevant for the discussion to follow, it can therefore safely be ignored from now on.

4.4.2 Composition scheme

Let T be the *LTS* of a high-level model M and let $\mathsf{Z}^T{<}\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}}{>}$ be the symbolic representation of T according to the definitions introduced in Sec. 3.5.1 (p. 57ff).

Theorem 4.5: *The composition scheme introduced in Eq. 4.11 delivers the superset of transitions* Z^p *of a high-level model* M*, so that*

$$\mathsf{Z}^T{<}\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}}{>} \subseteq \mathsf{Z}^p$$

For showing this, we will show how the activity-local structures as generated by the activity-local scheme can be deduced from the symbolic structure Z^T representing the high-level model's underlying transition system T.

Proof: Each transition is equipped with the label of the activity it was induced by, thus one may decompose Z^T into a sum of sets of transitions carrying the same label:

$$\mathsf{Z}^T = \sum_{l \in \mathcal{A}ct} \tilde{\mathsf{Z}}_l^C, \text{ where } \tilde{\mathsf{Z}}_l^C := p\mathtt{ZRestrict}(\mathsf{Z}^T, \vec{\mathsf{a}}, \mathcal{E}(l)). \tag{4.16}$$

In order to remove now the activity labels from the symbolically represented activity local transition systems, one may simply apply an existential abstraction on each $\tilde{\mathsf{Z}}_l^C$ for the boolean variables $\vec{\mathsf{a}}$:

$$\mathsf{Z}_l^C := p\mathtt{ZAbstract}(\tilde{\mathsf{Z}}_l^C, \vec{\mathsf{a}}, +) \text{ so that } \tilde{\mathsf{Z}}_l^C := \mathsf{Z}_l^C \times \mathsf{A}_l \tag{4.17}$$

where $\mathsf{A}_l := \mathsf{A}_{\{\vec{\mathsf{a}} := \mathcal{E}(l)\}}$ is the symbolic representation of activity label l. Since Z_l^C depends only on the remaining variables $\mathcal{V}^G \setminus \vec{\mathsf{a}}$ (cf. Eq. 4.10, p. 85), we can decompose each Z_l^C into a symbolic structure Z_l^D depending on the boolean variables of $\mathcal{V}^{\mathsf{D}_l}$ and into a symbolic structure Z_l^I depending on the boolean variables of $\mathcal{V}^{\mathsf{I}_l}$ (cf. Def. 4.9, p. 85). I.e. we have:

$$\begin{array}{l} \mathsf{Z}_l^D := p\mathtt{ZAbstract}(\mathsf{Z}_l^C, \mathcal{V}^{\mathsf{I}_l}, +) \\ \mathsf{Z}_l^I := p\mathtt{ZAbstract}(\mathsf{Z}_l^C, \mathcal{V}^{\mathsf{D}_l}, +) \end{array} \tag{4.18}$$

where Z_l^D is exactly the structure generated by the activity-local SG generation scheme. The operation $\mathsf{Z}_l^D \times \mathsf{Z}_l^I$ gives then the superset of transitions for each activity l: $\mathsf{Z}_l^C \subseteq \mathsf{Z}_l^D \times \mathsf{Z}_l^I$, since from $\mathcal{V}^{\mathsf{D}_l} \cap \mathcal{V}^{\mathsf{I}_l} = \emptyset$ it follows that $\mathsf{Z}_l^D \times \mathsf{Z}_l^I$ encodes the cross product of the Boolean vectors fulfilling the boolean function represented by Z_l^D, Z_l^I respectively (cf. Sec. 3.5.2, p. 62f). This result together with Eq. 4.16 and 4.17 yields:

$$\mathsf{Z}^T = \sum_{l \in \mathcal{A}ct} \mathsf{Z}_l^C \times \mathsf{A}_l \subseteq \sum_{l \in \mathcal{A}ct} \mathsf{Z}_l^D \times \mathsf{Z}_l^I \times \mathsf{A}_l \tag{4.19}$$

Now we will show that the satisfaction set of Z_l^I is a subset of the one of $\mathbb{1}_l$ ($Sat_{\mathbb{1}_l}$). This is intuitively clear, since Z_l^I contains only the markings of the independent SVs as contained within the reachable states from which a transition labeled with l emanates from, where the satisfaction set $Sat_{\mathbb{1}_l}$ of $\mathbb{1}_l$ contains $2^{|\mathcal{V}^{\mathsf{I}_l}|}$ elements, bit combinations resp. (cf. Def. 3.14, p. 61).
Let Ψ_l be now the set of all activity-independent markings as contained in a reachable state from which a transition with label l emanates from:

$$\Psi_l := \{\vec{s}_{I_l} | \vec{s}_{I_l} := \chi^I{}_l(\vec{s}) \wedge\ (\vec{s}, l, \vec{s}^{\,l}) \in T\} \tag{4.20}$$

Each SV $\mathfrak{s}_k \in \mathfrak{S}_l^{\mathcal{I}}$ does not change its value if activity l is executed (cf. Sec. 4.2.3, p. 83f). Consequently the i'th variable of $\mathcal{V}_{\mathsf{s}}^{\mathsf{I}_l}$ and $\mathcal{V}_{\mathsf{t}}^{\mathsf{I}_l}$ within a transition $\vec{s} \xrightarrow{l} \vec{t} \in T$ encode the same value. Thus Z_l^I represents the following function:

$$f_{\mathsf{Z}_l}(\mathsf{s}_1, \ldots, \mathsf{s}_n, \mathsf{t}_1, \ldots, \mathsf{t}_n) := \begin{cases} 1 \Leftrightarrow \mathcal{E}^{-1}(\vec{\mathsf{s}}) \in \Psi_l \wedge\ \eta(\mathsf{s}_i) = \eta(\mathsf{t}_i) \\ 0 \text{ else} \end{cases}$$

where $\eta(x)$ gives the value currently held by variable x. Thus f_{Z_l} is the identity function restricted to the vectors as contained in Ψ_l. Consequently $\mathsf{Z}_l^I \subseteq \mathbb{1}_l$ must hold.

Replacing Z_l^I with $\mathbb{1}_l$ in Eq. 4.19 yields:

$$\mathsf{Z}^T{<}\vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}}{>} \subseteq \sum_{l \in \mathcal{A}ct} \mathsf{Z}_l^D \times \mathbb{1}_l \times \mathsf{A}_l, \tag{4.21}$$

where the right part of this equation is the composition formula of the activity-local scheme as introduced in Eq. 4.11 (p. 89). ■

From the above equation one can conclude that the composition scheme may construct a larger transition system with respect to the original transition system T. However, as it will be shown in the next section, the additional transitions as contained in the set of composed transitions are not *false positives*, they emanate from unreachable states only.

4.4.3 Reachability analysis

Irrespective of the employed variant, symbolic reachability analysis starts with the initial state, where its symbolic representation serves as initial value to Z_R (cf. line 0 Algo. 4.1). The set of states to be explored in the next step is represented by *Mt*-DD Z_U, which is initialized with the set of states already known to be reachable (line 0 of Algo. 4.3.A and B, p. 90). For simplicity one may ignore the early-update strategy applied to Z_U within Algo. 4.3.B for the moment. Furthermore it is irrelevant that within Algo. 4.3.B the transition relations are applied to Z_U in a sequential manner (line 6-10), whereas Algo. 4.3.A follows an all-at-once strategy (line 3). The execution of the set of potential transition functions, in the states stored in Z_U yields the one-step reachability set (Z_{tmp}) of all transitions emanating from a state $\vec{s} \in \mathsf{Z}_U$. We will now prove that the additional transitions contained in the set of composed states, denoted Z^p and as constructed in Eq. 4.21 (p. 97) can only emanate from non-reachable states and that the symbolic reachability algorithms construct therefore the complete and correct sets of reachable states.

Let the symbolic structure Z_R encode the set of reachable states and let the symbolic structure Z_T encode the set of transition functions of a high-level model M. Let Z_T^p be the potential transition system as derived from M by applying the activity-local scheme. Let a false positive be a transition $\mathbf{f} := (\vec{s}, l, \vec{t})$ where $\vec{s} \in \mathsf{Z}_R \wedge\ f \notin \mathsf{Z}_T$ holds.

Lemma 4.6: Z_T^p *contains no false positive.*

The proof of the above lemma will be carried out to by constructing a contradiction.

Proof: Assumption: Z_T^p contains a false positive $\mathbf{f}$.
Since $\vec{s} \in \mathsf{Z}_R$, there must be a finite sequence of activity executions, leading from the initial state $\vec{s}^{\epsilon}$ to state $\vec{s}$ (repetitive execution of line 3 - 7 of Algo. 4.3.A and line 5-11 of Algo. 4.3.B). This execution sequence can be generated, irrespective if one successively executes Z_T or the potential transition function Z_T^p, since as shown before $\mathsf{Z}_T \subseteq \mathsf{Z}_T^p$ holds. This sequence of activity executions visits reachable states only, which is intuitively clear, but can also be shown via induction over the length of activity execution sequences as constructed by repetitively executing the (outer) DO-UNTIL loop of Algo. 4.3.A and B.
Consequently the execution of the potential transition function Z_T^p in a state $\vec{s}$ must give then the false positive $(\vec{s}, l, \vec{t})$, but where $\vec{s}_{I_l} = \vec{t}_{I_l}$ must hold! It follows immediately that $(\vec{s}, l, \vec{t})$ must be a valid state change for the following reasons: (a) The values stored on the positions of $\mathfrak{S}_l^{\mathcal{T}}$ are valid values, since they are contained in the reachable state $\vec{s}$. (b) The change of the values of position referring to the SVs of $\mathfrak{S}_l^{\mathcal{D}}$ is also correct, since it was obtained from explicit exploration. *This gives that* $\vec{s}, \vec{t} \in \mathsf{Z}_R$ *and therefore also* $\mathbf{f} \in \mathsf{Z}_T$ *must hold. This is a contradiction of the initial assumption, which stated that* $\mathbf{f}$ *is a false positive and contained in* Z_T^p. ■

Due to the above lemma it is clear that the additional transitions as contained in Z_T^p can solely emanate from unreachable states. Based on this we can now show that the Algo. 4.3.A and B compute the correct set of reachable states Z_R.

Theorem 4.7: *Given a potential transition system* Z_T^p*, which is the superset of transitions of a transition system* T *and given the initial system state*[6] *as input, the symbolic reachability Algo. 4.3.A and B generate a complete and correct symbolic representation of* T*'s set of reachable states.*

Proof: As shown in the previous section $\mathsf{Z}_T \subseteq \mathsf{Z}_T^p$ holds. According to the above lemma Z_T^p does not contain any false positives, so that the repetitive application of Z_T^p to the states of Z_U yields reachable states only, which are stored in the symbolic structure Z_R (line 6 in Algo. 4.3.A and line 5 in Algo. 4.3.B). Since the number of transitions stored in Z_T^p is finite, the repetitive execution of Z_T^p to the states of Z_U will reach a fixed point and Algo. 4.3.A

[6] As initial state one can employ any state from which all other states of T are reachable.

and B terminate. Since according to Theorem 4.5 Z_T^p contains all reachable transitions, the complete set of reachable states has been generated then, which is due to the fact that

$$\mathsf{Z}_R \subseteq \mathsf{Z}_R^p \text{ where } \mathsf{Z}_R^p := p\mathtt{ZAbstract}(\mathsf{Z}_T^p, \mathcal{V}_\mathsf{t}, +) + p\mathtt{ZAbstract}(\mathsf{Z}_T^p, \mathcal{V}_\mathsf{s}, +)\{\vec{\mathsf{s}} \leftarrow \vec{\mathsf{t}}\}$$

and the remaining states as contained in Z_R^p are not reachable. Thus the generated symbolic structure Z_R is a complete and correct representation of the set of reachable states of a high-level model's underlying *sLTS* T. ■

This concludes our discussion on the completeness and correctness of the activity-local scheme.

4.5 Computing performability measures

For obtaining performability measures of the system under study the modeler specifies PVs on the level of the high-level model description. As pointed out in Sec. 4.2.2 a PV consists of a set of rate rewards - and / or impulse reward functions and a type. The set of all reward functions specified on the high-level model was defined to be denoted $\mathcal{R}$ for the rate and $\mathcal{I}$ for the impulse rewards. If for all elements of $\mathcal{R}$ and $\mathcal{I}$ a symbolic representation was generated, one simply needs to combine them via summation, in order to generate symbolic representations of the user-defined PVs, which are in fact aggregated rate and impulse reward values ($\mathcal{R}_p$ and $\mathcal{I}^p$ of Def. 4.17, p. 81). The first and second moments of PVs can then be computed via a graph-traversing algorithm, given that the transient or steady-state probabilities have been computed before. However, depending on the fact whether one intends to compute instant-of-time, interval-of-time or time averaged interval-of-time values, a multiplication of the result with $\triangle t$ must follow, where in case of instant-of-time and time averaged interval-of-time performance measures we define $\triangle t := 1$ (cf. Sec. 2.2.2, p. 15ff).

The above illustrated functionality is incorporated into algorithm `ComputePV` (Algo. 4.5) and its sub-routines. It employs the following data structures:

(1) Z_R representing the set of reachable states and Z_T representing the set of reachable transitions.

(2) Z_R^o the offset-labeled z-BDD representing the set of reachable states. This structure is needed for determining the (dense) state indices, while traversing the symbolically represented reward functions and computing the performability measures.

(3) *prob* the vector of state probabilities (transient or steady state).

(4) The current PV p which is assumed to additionally offer substructures for storing the moments and variance of $\mathcal{R}_p$ and $\mathcal{I}^p$.

(5) Z_{rate}^p and Z_{imp}^p, the symbolic structures representing $\mathcal{R}_p$ and $\mathcal{I}^p$ of the current PV p.

After executing line 0-4 the offset-labeled z-BDD representing the set of reachable states, and the symbolic representations of each reward function is given, where the routines `MakeImpulseRewards` and `MakeRateRewards` will be explained below, for details on offset-labeling the reader may refer to Sec. 3.5.3 (p. 66ff). Now one is ready to compute the user-defined performability measures, which is done in the FOR-loop (line 5-17, Algo. 4.5). As first step one computes either transient or steady-state probabilities, depending on the type of the current PV (line 6). In the next step one merges the individual rate - and impulse reward functions for each PV p in order to construct a symbolic representation of $\mathcal{R}_p$ and $\mathcal{I}^p$ (line 7 and 8), where subsequently the root nodes of the resulting *Mt*-DDs, as well as the root node of the offset-labeled BDD Z_R^o are fetched (line 9-11). These nodes, as well as the memory locations for storing the first and second moment of a PV's rate and impulse function is passed to algorithm `ComputeMoment`. This algorithm computes first and second moments

Algorithm 4.5 Main routine for computing user-defined PVs

```
ComputePV()
(0)  Z_R := ExploreStateGraph();
(1)  Z_T = Z_T · Z_R;
(2)  Z_R^o := OffsetLabel(Z_R);
(3)  MakeRateRewards(Z_R);
(4)  MakeImpulseRewards(Z_R);
(5)  FOR p ∈ PV DO
(6)     prob := ComputeStateProbabilities(p.type, Z_R^o, Z_T);
(7)     Z_rate^p := Σ_{r∈R^p} R_r;
(8)     Z_imp^p := Σ_{i∈I^p} I^i;
(9)     n := getRoot(Z_rate^p), r := getRoot(Z_R^o);
(10)    ComputeMoment(n, r, 0, p.r_mean, p.r_var);
(11)    n := getRoot(Z_imp);
(12)    ComputeMoment(n, r, 0, p.i_mean, p.i_var);
(13)    p.r_mean := p.r_mean · △t, p.r_var := p.r_var · △t^2;
(14)    p.i_mean := p.i_mean · △t, p.i_var := p.i_var · △t^2;
(15)    p.r_var := p.r_var − p.r_mean^2;
(16)    p.i_var := p.i_var − p.i_mean^2;
(17) END
(18) RETURN ;
```

of symbolically represented reward functions, given that the symbolic representation of the set of reachable states is offset-labeled and the state probabilities are stored in an array of respective size. In case one intends to compute interval-of-time measures the obtained results must be normed to the length of the interval ($\triangle t$) or its square ($\triangle t^2$), which is done in line 13 and 14. – In case of instant-of-time and time-average intervals-of-time measures one simply defines $\triangle t$ to be 1.– Based on this, algorithm `ComputePV` computes the variance for $\mathcal{R}_p$ and $\mathcal{I}^p$ (line 15 and 16). Once all PVs are processed algorithm `ComputePV` terminates and the desired performability measures for the system under study are computed. In the following we will now briefly discuss the routines as employed by the main algorithm `ComputePV`.

4.5.1 Computing state probabilities

By executing one of the iterative solution method as described in Sec. 2.2.2, one can compute steady state and transient state probabilities for a given high-level MRM. In this thesis we employ their hybrid implementation as already illustrated in Sec. 3.5.3 (p. 66ff). In contrast to pure symbolic solvers the hybrid ones employ solely a symbolic representation for the transition rate matrix, whereas the matrix diagonal of the latter, as well as the iteration vectors are stored as plain arrays. At termination the hybrid solver delivers the vector *prob*, containing either transient state or steady state probabilities for each state. What follows next is the generation of the symbolic representation of the user-defined reward functions.

4.5.2 The reward-local scheme: Generating representations of reward functions

Traditionally one computes PVs or their rate and impulse rewards while generating the high-level model's SG. However, under symbolic SG representation only a fraction of states is visited explicitly. Given that a reward returning function may be of arbitrary complexity, but solely depends on a subset of $\mathfrak{S}$, it seems reasonable, to compute them once the symbolic representation of the overall SG is generated. In order to explicitly process as few states as possible we once again consider only local information of the reward function definitions (cf. Def. 4.15 and 4.16). I.e. in case of rate reward r it is assumed that a rate reward specific set of dependent SVs $\mathfrak{S}_r^{\mathcal{D}}$ is present. Analogously to activity-local markings one will speak of rate reward-local markings, when addressing the positions referring to elements of $\mathfrak{S}_r^{\mathcal{D}}$ in a model's state $\vec{s}$. In case impulse reward functions, it is assumed that all impulse rewards as induced by an activity k solely depend on $\mathfrak{S}_k^{\mathcal{D}}$. The above illustrated ideas are incorporated

Algorithm 4.6 Generating symbolic representations of reward functions

`MakeRateRewards`()
(0) FOR $k \in \mathcal{R}$ DO
(1) $\mathsf{R}_k := \emptyset$;
(2) $\mathsf{Z}_U := p\texttt{ZAbstract}(\mathsf{Z}_R, \mathcal{V}_{\mathsf{s}}^{\mathsf{I}_k}, +)$;
(3) WHILE $\mathsf{Z}_U \neq \emptyset$ DO
(4) $\mathsf{Z}_s \xleftarrow{\vec{s}_{D_k}} \mathsf{Z}_U$;
(5) $\mathsf{Z}_U := \mathsf{Z}_U \setminus \mathsf{Z}_s$;
(6) $\mathsf{Z}_{tmp} := \mathsf{Z}_R \cdot \mathsf{Z}_s$;
(7) $\mathsf{Z}_s \xleftarrow{\vec{s}} \mathsf{Z}_{tmp}$;
(8) $\vec{s} := \mathcal{E}^{-1}(\mathsf{Z}_s)$;
(9) $rew := \mathcal{R}_k(\vec{s}_{D_k})$;
(10) IF $(rew \neq 0)$ THEN
(11) $\mathsf{R}_k := \mathsf{R}_k + rew \cdot \mathsf{Z}_{tmp}$;
(12) END
(13) END
(14) RETURN ;

(A) Obtaining a symbolic representation-for each rate returning function

`MakeImpulseRewards`()
(0) FOR $i \in \mathcal{I}$ DO
(1) FOR $k \in \mathcal{A}ct_i^m$ DO
(2) $\mathsf{I}_k^i := \emptyset$;
(3) $\mathsf{Z}_U := p\texttt{ZAbstract}(\mathsf{Z}_k, \mathcal{V}_{\mathsf{t}}^{\mathsf{D}_k}, +)$;
(4) WHILE $\mathsf{Z}_U \neq \emptyset$ DO
(5) $\mathsf{Z}_s \xleftarrow{\vec{s}_{D_k}} \mathsf{Z}_U$;
(6) $\mathsf{Z}_U := \mathsf{Z}_U \setminus \mathsf{Z}_s$;
(7) $\mathsf{Z}_{tmp} := \mathsf{Z}_R \cdot \mathsf{Z}_s$;
(8) $\mathsf{Z}_s \xleftarrow{\vec{s}} \mathsf{Z}_{tmp}$;
(9) $\vec{s} := \mathcal{E}^{-1}(\mathsf{Z}_s)$;
(10) $imp := \mathcal{I}_k^i(\vec{s}_{D_k})$;
(11) IF $(imp \neq 0)$ THEN
(12) $\mathsf{I}_k^i := \mathsf{I}_k^i + imp \cdot \mathsf{Z}_{tmp}$;
(13) END
(14) END
(15) $\mathsf{I}^i := \mathsf{I}^i + \mathsf{I}_k^i$;
(16) END
(17) RETURN ;

(B) Obtaining a symbolic representation-for each impulse returning function

into the algorithms `MakeImpulseRewards` and `MakeRateRewards` (Algo. 4.6). These are the algorithms for generating symbolic representations of rate - and impulse reward functions and which will be explained now.

Generating representations of rate reward functions

Algorithm `MakeRateRewards` (Algo. 4.6.A) consists of two nested loops. In the outer FOR-loop one processes each rate reward function as defined by the user, whereas in the inner WHILE-loop these definition are processed individually. I.e. at first the set of reachable states is reduced to reward-local markings by simply abstracting from those boolean variables referring to the rate reward's set of independent SVs ($\mathfrak{S}_r^{\mathcal{I}} := \mathfrak{S} \setminus \mathfrak{S}_r^{\mathcal{D}}$). From the obtained set of rate reward-local markings represented by symbolic structure Z_U on pops now "*rate-local*" marking by "*rate-local*" marking and temporarily stores each in the symbolic structure Z_s (line 4). The set of reachable states which are all equivalent concerning the positions of $\mathfrak{S}_r^{\mathcal{D}}$ is then generated in line 6 and stored within symbolic structure Z_{tmp}. Now one simply extracts one of the states as contained within Z_{tmp} and calculates its rate reward concerning rate reward function r, which is done by explicitly executing r's specific reward returning function (line 7 - 9). In case this reward is not equal to 0, one simply needs to multiply the symbolic structure Z_{tmp} with this constant and add the result to the the previously computed pairs of states and rewards as represented by R_k (line 10 and 11). This procedure is repeated until all "*rate-local*" markings are processed and Z_U represents the empty set (cf. line 5). Rather than processing all states for each rate reward, algorithm `MakeRateRewards` handles classes of states per step, where each equivalence class consists of all states being equivalent concerning the values held by r's dependent SVs ($\mathfrak{S}_r^{\mathcal{D}}$).

For exemplification, one may assume that rate reward r is defined as the number of tokens contained in place p_5 of the SPN of Fig. 4.1.A (p. 92). The symbolic structure (here a ZDD) Z_R of Fig. 4.4 (p. 102) encodes the set of reachable states, which is the initial value of Z_U. Let us further assume that the states popped from the symbolic structure Z_U in the inner for loop are the following: (00 00 2), (00 01 1) and (01 01 0). The corresponding pZDD Z_s and ZDD Z_{tmp}, as obtained after executing line 4 and 6 of algorithm `MakeRateRewards` for the three states are then also depicted in Fig. 4.4. The final symbolically encoded rate reward function obtained at termination is given as the ZDD R_r. Rather than computing

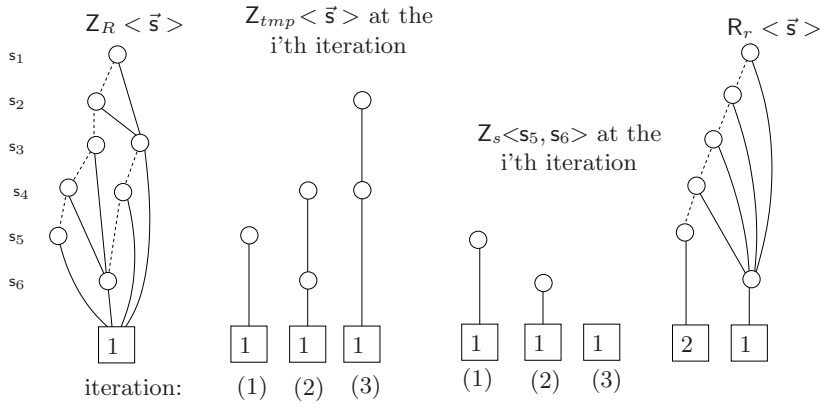


Figure 4.4: Exemplification of the reward-local approach

and encoding the rate reward r for each of the 9 states as contained in Z_R, this is only done 3 times, namely once for the states where $p_5 = 2$, once for the states where $p_5 = 1$, and once for the states where $p_5 = 0$.

Generating representations of impulse reward functions

Algorithm `MakeImpulseRewards` specified as Algo. 4.6.B is organized in a similar way as the algorithm for generating symbolic representations of rate reward functions. But since an activity may generate different impulse rewards for different impulse reward functions, one needs to iterate over three nested loops. In the outer two FOR-loops one processes each impulse reward function and its respective sets of activities. Within the inner WHILE-loop one processes all activity-local markings the current activity k is enabled in (line 5-12). The impulse reward as induced by the activity's execution is calculated in line 10. In case it is not equal to 0, one simply multiplies the symbolic structure Z_{tmp} with this value. This gives one a symbolic representation of all states being equivalent concerning the current activity-local marking $\vec{s}_{D_k}$, where these states of course have all the same impulse reward value with respect to activity k and impulse i. Due to the construction of Z_U, Z_s and thus Z_{tmp}, the obtained pairs of states and impulse rewards are automatically weighted by the execution rate of the activity under process, so that Def. 4.16 (p. 80) is satisfied. At termination of the inner FOR-loop (line 1 -14) one ends up with a symbolic representation for each activity-specific impulse reward function with respect to impulse reward i. In line 15 one finally builds iteratively the sum over all symbolic structures encoding such activity-specific impulse reward functions, so that one obtains a symbolic representation for each impulse reward function i as contained within a user-defined PV.

4.5.3 Computing moments of performance variables

Algorithm `ComputeMoment` (Algo. 4.7) computes first and second moment of the reward functions by simultaneously traversing the symbolic structure representing the reward function and the offset-labeled structure representing the set of reachable states. As input parameters this algorithm takes the root nodes of the respective symbolic structures, the initial offset value off, which is set to 0 and the variables for storing first and second moment of the current reward function. While traversing the symbolic structures the state index of the traversed path is obtained by summing over the offsets of nodes left via `then`-edge (line 6). In case one reaches a terminal non-zero node, the index of the current state is known and one can successively compute mean and second moment of the respective reward (line 1 -2). The vector *prob*, which holds steady state or transient state probabilities is hereby assumed to be globally visible.

Algorithm 4.7 Algorithm for computing moments of PVs via graph-traversal

```
ComputeMoment(node1, node2, off, m, v)
(0)  IF n ∈ K_T THEN
(1)      m := m + prob[off] * value(node1);
(2)      v := v + prob[off] * value(node1)^2;
(3)  ELSE IF var(node1) π> var(node2) THEN
(4)      ComputeRew(node1, else(node2), off, m, v);
(5)  ELSE
(6)      ComputeRew(then(node1), then(node2), r.offset + off, m, v);
(7)      ComputeRew(else(node1), else(node2), off, m, v);
(8)  RETURN ;
```

4.6 Extending the basic activity-local scheme

In this section we will cover issues related to the handling of (a) user-defined symmetries within the SG, and (b) models, where immediate activities are present.

4.6.1 Handling explicitly modeled symmetries

For mapping a CTMC to its reduced (bisimilar) counterpart, generally applicable, symbolic algorithms are known [Sie01]. However, non- and symbolic approaches are also known to be not very efficient, since partitioning the state space without having apriori knowledge of the high-level model-structure, turns out to be cumbersome in practice. As simplification [San88, Sie95] therefore suggest the construction of high-level models by employing the *Rep*-operator, which induces submodel-imposed symmetries on the SG of the overall model and thus enables a partitioning of the state space, where the individual state of the partitions are strictly lumpable. By taking advantage of the explicitly defined symmetric model structure, the reduced SG of the overall model can in principle be constructed on-the-fly, i.e. during explicit SG generation (cf. Sec. 2.2.3, p. 17 and Sec. 2.4, p. 24). However, in the context of the activity-local scheme a symbolic implementation of such an on-the-fly procedure seems to destroy the partial character of the explicit SG generation. Therefore it seems more appropriate to generate first the un-reduced SG and subsequently apply the state lumping on the level of the symbolic structure, especially since the storage of the un-reduced SG is also not problematic. By taking advantage of the symmetric model structure, a procedure to be designed, is expected to be more efficient as the general, symbolic approach introduced in [Sie02].
Since the ideas developed below also applies to Markov chains consisting of tangible and vanishing states, we will now generically speak of rates, rather than always referring also to the aggregation of execution probabilities of symmetric activities.

For exemplification one may turn now once again to Fig. 4.1.A (p. 92). Instead of a monolithic model the SPN depicted there shall be organized in a structured manner. To do so we build 3 non-disjoint submodels: The first submodel M_1 consists of the place p_1, p_3, p_5 and the activities a and b, the second submodel M_2 consists of the places p_3, p_4, p_5 and the activities c and d. The remaining activity e and its set of dependent places builds submodel M_3. The submodels are composed via the sharing of places having the same label within the submodel, i.e. the places p_1, p_2 and p_5 are the shared ones among the submodels. In order to make use of a user-defined symmetry, it is specified that submodel M_2 is replaced with another instance of submodel M_1. This process is depicted in Fig. 4.5 (p. 104). Let each submodel-local state descriptor be organized as follows:

- lower instance of M_1^l: $\vec{s}^l := (p_1^l, p_2^l, p_5)$
- upper instance of M_1^u: $\vec{s}^u := (p_1^u, p_2^u, p_5)$ and
- instance of model M_3: $\vec{s}^{M_3} := (p_1^l, p_1^u, p_5)$.

If the joined state variables are mapped to the same vector component a state may be describe by vector $\vec{s} := (p_1^l, p_2^l, p_1^u, p_2^u, p_5)$. Via the activity-local SG generation scheme one

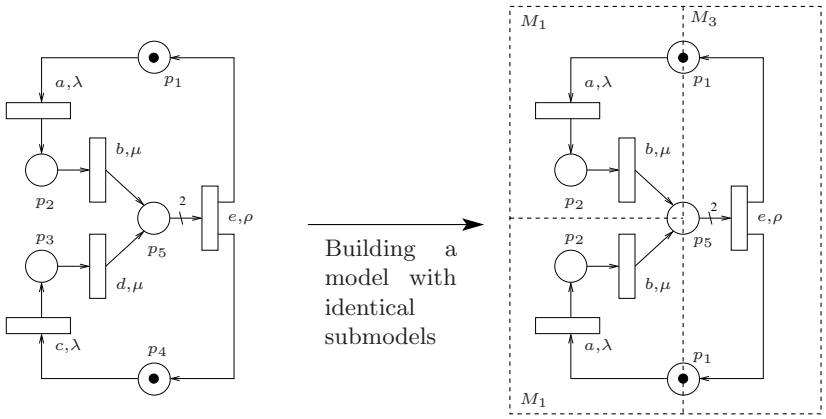


Figure 4.5: SPN with user-defined symmetric submodels

may now construct the un-reduced SG, which we already depicted in Fig. 2.1.A (p. 19). For applying the state lumping, all states which are permutations of each other, with respect to the identical (or symmetric) submodels, are replaced by their representative or macro state Thus analogously to the procedure illustrated at end of Sec. 2.4 (p. 24f) states being submodel-wise permutations from each other, e.g. $((0,1),(1,0),p_5)$ and $(1,0),(0,1),p_5)$ are known to be strictly lumpable. In Fig. 2.1.A these (symmetric) states were marked with the same number, i.e. the number of their equivalence class. If these states are lumped, i.e. replaced by their macro state and the rates are adapted accordingly (cf. Sec. 2.2.3), one obtains the lumped SG as illustrated in Fig. 2.1.B. Since each partition is solely represented by its macro state, the multiply occurring transitions among the macro states and the states of the other partitions are aggregated via summation, which is the strategy already illustrated in Sec. 2.2.3 and which is the strategy behind algorithm `LumpSG` (Algo. 4.8), which we will explain in detail now.

Let the sub-vector, describing the state of all symmetric submodels be denoted as replica-local vector. For realizing the lumping of states a posteriori to the construction of a symbolic representation of the un-reduced SG, one generates first a symbolic representation for each partition of strictly lumpable states. Therefore the boolean `s`- and `t`-variables encoding the SVs of all identical (symmetric) submodels are defined as follows: $\mathcal{V}^R := \mathcal{V}_\mathsf{s}^R \cup \mathcal{V}_\mathsf{t}^R$, where the elements of $\mathcal{V}_\mathsf{s}^R$ hold the values of the replica-local source states and the elements of $\mathcal{V}_\mathsf{t}^R$ hold the values of the replica-local target states. In the upper FOR-loop of algorithm `LumpSG` one computes a replica-local symbolic representation for each partition $\mathsf{Z}_{sub}^i <\mathcal{V}_\mathsf{t}^R>$ and already chooses a replica-local macro state $\mathsf{R}^i <\mathcal{V}_\mathsf{s}^R>$ (line 1-7). This means in detail that in line 2 one simply pops an arbitrary state from the set of the replica-local states, and constructs all of its submodel-wise permutations, stored as target states, which is done by calling function `MakeAllPerm`() (line 3). In line 4 one chooses a state descriptor for representing the macro state (R_{sub}^i), whereas in line 5 all states referring to the current partition of replica-local states are removed from the set of all replica-local states. The upper FOR-loop terminates as soon as the set of replica-local states is empty. Thus at termination each partition has its symbolic representation, as well as its symbolic represented macro state.
As next step one needs to compute the cumulative rates and construct the reduced SG, where we follow the idea illustrated above. Let N_C be now the number of partitions as known after executing the upper FOR-loop. Within the inner FOR-loop (line 9-13) one first extracts all transitions emanating from states carrying the replica-local marking of the current macro state within the variables of $\mathcal{V}_\mathsf{t}^R$ (line 10). In line 11 we simply remove the positions referring to the replica-local markings, where the rates of collapsing transitions are automatically summed. As next the reduced SG is iteratively generated by simply filling the empty positions of Z_{tmp} with the replica-local marking of the macro state representing the target partition and subsequently adding this new set of transitions to the previously computed ones (line 12). This procedure is repeated until all partitions have been processed (line 8

Algorithm 4.8 Applying the lumping theorem in case of user-defined model-symmetries

LumpSG()

(0) $\mathsf{Z}_{sub}<\mathcal{V}_\mathsf{s}^R>:= p\mathtt{ZAbstract}(\mathsf{Z}_R, \mathcal{V}_\mathsf{s} \setminus \mathcal{V}_\mathsf{s}^R, +);$

/* *Generate equivalence classes on the level of symmetric submodels* */

(1) FOR $(N_C := 0, i := 1;\ \mathsf{Z}_{sub} \neq \emptyset;\ N_C{+}{+}, i{+}{+})$ DO

(2) $\quad \mathsf{Z}_{\vec{s}}<\mathcal{V}_\mathsf{s}^R> \xleftarrow{\vec{s}} \mathsf{Z}_{sub};$

(3) $\quad \mathsf{Z}^i_{sub}<\mathcal{V}_\mathsf{t}^R>:= \mathsf{MakeAllPerm}(\mathsf{Z}_{\vec{s}});$

(4) $\quad \mathsf{R}^i_{sub}<\mathcal{V}_\mathsf{s}^R>:= \mathsf{GetClassRep}(\mathsf{Z}^i_{sub});$

(5) $\quad \mathsf{Z}_{sub}<\mathcal{V}_\mathsf{s}^R>:= \mathsf{Z}_{sub} \setminus \mathsf{Z}^i_{sub}\{\mathcal{V}_\mathsf{s}^R \leftarrow \mathcal{V}_\mathsf{t}^R\};$

(6) END

/* *Compute cumulative probabilities and rates, and generate reduced SG* */

(7) $\mathsf{Z}'_T <\mathcal{V}_\mathsf{s}, \mathcal{V}_\mathsf{t}>:= \emptyset;$

(8) FOR $(i := 1; i \leq N_C; i{+}{+})$ DO

(9) $\quad$ FOR $(j := 1; j \leq N_C; j{+}{+})$ DO

(10) $\quad\quad \mathsf{Z}_{tmp}<\mathcal{V}_\mathsf{s}, \mathcal{V}_\mathsf{t}>:= \mathsf{Z}_T \cdot \mathsf{R}^i_{sub}<\mathcal{V}_\mathsf{s}^R> \cdot \mathsf{Z}^j_{sub}<\mathcal{V}_\mathsf{t}^R>;$

(11) $\quad\quad \mathsf{Z}_{tmp} := p\mathtt{ZAbstract}(\mathsf{Z}_{tmp}, \mathcal{V}_\mathsf{t}^R, +);$

(12) $\quad\quad \mathsf{Z}'_T := \mathsf{Z}'_T + \mathsf{Z}_{tmp} \times \mathsf{R}^j_{sub}<\mathcal{V}_\mathsf{t}^R>;$

(13) $\quad$ END

(14) END

(15) $\mathsf{Z}_T := \mathsf{Z}'_T;$

(16) $\mathsf{Z}_R := p\mathtt{ZAbstract}(\mathsf{Z}_T, \mathcal{V}_\mathsf{s}, +) + p\mathtt{ZAbstract}(\mathsf{Z}_T, \mathcal{V}_\mathsf{t}, +);$

(17) RETURN ;

-14). Thus the complexity of the above algorithm is $O(N_C^2)$, since one needs to compute the cumulative rates leading from partition i to partition j for each partition separately.

It is important to note that the above illustrated procedure assumes knowledge about the submodels behaving symmetric to each other, as it is the case for the *Rep*-composition operator (cf. Sec. 2.3.3, p. 23). However, in case the model contains multiple instances of the same submodel, it is required that the user specifies, if they behave symmetric or not, which of course depends on the environment. This is already illustrated by the example of Fig. 4.5. Here states being submodel-wise permutations from each other, e.g. $((0,1),(1,0),p_5)$ and $(1,0),(0,1),p_5)$ belong to the same partition, even though the respective submodel-local states with respect to submodel M_3 differ ($(1,0,p_5)$ and $(0,1,p_5)$). But this does not matter, since the submodel local behavior of M_3 is identical for both states, so that states of the kind $(p_1^l, p_2^l,), (p_1^u, p_2^u), p_5)$, where the first tuples are permutations of each other belong to the same partition of strictly lumpable states. As a consequence, it is clear that the identical submodels M_1^l and M_1^u only exhibit symmetric behavior, since the environment is reacting with them in the same way. On the other hand it makes also clear, that an on-the-fly procedure as illustrated in Sec. 2.4, as well as the approach presented here can only be applied, if the user-explicitly defines that M_1^l and M_1^u, will have the same behavior. In case of the *Rep*-operator, where either SVs or activities are shared with the environment, and among all replicas, this is implicitly guaranteed. In case this symmetry is not explicitly specified or imposed by the composition operator, submodel-imposed symmetries can only be detected by applying a standard symbolic algorithm for SG reduction (cf. [Sie02]). However, as already mentioned above, these algorithms are known to be computational expensive, but deliver the smallest equivalence relation, i.e. the one with the a minimal number of partitions. –We will come back to this issue in Sec. 6.3.

The symbolic representations of reduced SGs are known to be less memory efficient, if compared to their un-reduced counter parts. However, this phenomenon is not surprising, since the number of rates among the states increases, whereas the number of represented transition decreases. Thus the symbolic structure may benefit less from the isomorphism-reduction rule (cf. Sec. 3.2.1, p. 33ff), which is the most influential factor of keeping symbolic structures compact. However, in case of the hybrid solution method (cf. Sec. 4.5.1, p. 100) for computing state probabilities, the number of elements of the probability vector gives the

bottleneck. Thus an increased size of a symbolic representation of the transition rate matrix is of minor interest, whereas the reduced number of states plays a key role.

4.6.2 Handling of immediate activities

The activity-local scheme follows a selective *bfs* scheme for explicit SG exploration, where only local knowledge about activities is taken into consideration. However, this turns out to be problematic in case of immediate activities for the following reason: Let $l, k \in \mathcal{A}ct^i$ and let $(k, l) \in \mathcal{A}ct^{\mathcal{I}}$ hold. Now we define that $\exists \vec{x}, \vec{y} \in \mathbb{S} : \vec{x} \triangleright \{l, k\}; \vec{y} \triangleright \{l, k\}$; where $prio_l(\vec{x}_{D_l}) > prio_k(\vec{x}_{D_k}) \wedge prio_l(\vec{y}_{D_l}) < prio_k(\vec{y}_{D_k})$ holds. As a consequence activity l will sometimes suppress the execution of k, and sometimes k will suppress the execution of l. If one solely considers local information only and therefore executes k and l independently from each other, the explicit generation would yield the generation of false positives, where the target state of such a transition would wrongly be considered as reachable (cf. Sec. 4.4.3, p. 98). But not enough, the exploration of the target state of a false positive lead to other non-valid states, so that a previously bounded SG may become now unbounded. It is immediately clear that a correction of such false positives, a posteriori to SG generation, is therefore not possible. From this one may also conclude that immediate activities, which mutually suppress their execution are required to be tested always together, so that their enabledness is determined correctly. Such a strategy will be referred to as combined exploration strategy. The enforcement of a combined exploration scheme is straight forward, one simply defines that the dependent SVs, of all interfering activities are added to an activities set of dependent SVs. Unfortunately one can not decide *a priori* to SG generation whether $\exists l, k \in \mathcal{A}ct^i : l \notin \mathcal{A}^{D_k}$ and $\exists \vec{s} \in \mathbb{S} : \vec{s}\,[> l$, and $\vec{s} \triangleright k$ but $\vec{s}\,[\not> k$, since the reachability of $\vec{s}$ is semi-decidable only. Thus it is not possible to generally decided whether a `GS-SA net` requires the above re-definitions or not and which activities need to be grouped together. Therefore one is forced to re-define the sets of dependent SVs for each immediate activity in a brute-force manner:

Definition 4.28: Re-definition of the set of dependent SV of immediate activities:

The set of activity-dependent SVs for each $l \in \mathcal{A}ct^i$ is defined as follows:

$$\widetilde{\mathfrak{S}_l^{\mathcal{D}}} := \bigcup_{\forall k \in \mathcal{A}ct^i} \mathfrak{S}_k^{\mathcal{D}}$$

In the following we will denote the above set $\mathfrak{S}_{\mathcal{A}ct^i}^{\mathcal{D}}$, since $\widetilde{\mathfrak{S}_l^{\mathcal{D}}} = const$ for all $l \in \mathfrak{S}_{\mathcal{A}ct^i}^{\mathcal{D}}$.

Unfortunately it is directly clear that the above definition lowers the partial character of the explicit SG exploration of the activity-local approach. But on the other hand immediate activities can be considered as a kind of "syntactic sugar", since one can in principle eliminate them already on the level of the high-level model. Thus it seems reasonable to assume, that high-level model consists mainly of Markovian activities, rather than immediate ones. Thus the overhead imposed by a combined explicit exploration strategy for immediate activities should be small in practice.

In case the immediate activities have all the same constant priority $\neq 0$, the above re-definition of the activity-specific sets of dependent SVs of the immediate activities is obsolete. Since here immediate activities are not in the position to mutually suppress their enabledness. Consequently in such a setting the basic definition for a set of activity-dependent SVs (cf. Def. 4.2, p. 74) for each immediate activity is sufficient, but of course employed with the extended algorithms to be discussed next.

Explicit SG Generation and Encoding

Due to the above construction in combination with the sophisticated rule for deciding whether an activity is enabled or not (cf. Eq. 4.3 , p. 76 in combination with Def. 4.13, p. 79) algorithms `ExploreStates` and `EncodeTransitions` need to be slightly modified.

Explicit exploration of states

In case of exploration one simply tests, if there are immediate activities in the set of executable dependent activities (line 3-7 of Algo. C.1, p. 165). If this is the case one executes them and inserts the resulting transitions into the $\mathbb{T}$-Buffer. One may note that we directly compute the individual execution probabilities (line 5). This is justified, since all the immediate activities are explored together. In case there are no immediate activities to be executed, one explores the Markovian activities as before.

Encoding and testing for further exploration

The modified algorithm for encoding the established transitions and testing for further exploration is given as Algo. C.2 (p. 165). Since immediate activities suppress the enabling of Markovian activities, a newly reached state must first be tested for being vanishing or tangible, which is done in line 4 to 7. In this context line 5 also means that there exists no other immediate activity, which can suppress the enabledness of the current immediate activity handled. If there are no immediate activities enabled one continues with the Markovian activities as before (line 8 - 11).

Symbolic Composition

Even under a combined exploration strategy for the immediate activities as illustrated above, the scheme for symbolic composition may give one false positives, namely for $m \in \mathcal{A}ct^m, k \in \mathcal{A}ct^i$ and $(k,m) \in \mathcal{A}ct^{\mathcal{I}}$ in cases where $\vec{x}_{D_m} = \vec{y}_{D_m}$ and $\vec{x}\,[> m$ but $\vec{y}\,[\not> m$ since $\vec{y}\,[> k$. However, this problem can be fixed on the level of symbolic reachability analysis, since such false positives only appear on the level of symbolically represented transitions.

Symbolic reachability analysis

Since immediate activities suppress the enabling of Markovian activities having concession in the same state, it is clear that for the execution of the latter, it is essential that no immediate activity is enabled in the current state. This issue was solved on the level of explicit exploration by considering immediate activities first. However, symbolic composition may introduce false positives concerning the Markovian transitions, i.e. the scheme will generate states, where immediate and Markovian transitions may emanate from. For fixing this problem, we simply need to decide, whether a state is vanishing or not, before we decide whether the execution of the Markovian activities in this state must be prevented or not. Therefore the newly reached target states within the symbolic reachability analysis must first be tested if they are vanishing or not. Consequently the strategy of updating the set of unexplored states as soon as possible (early update strategy of Sec. 4.3.4, p. 90), can only be employed when exploring immediate activities. In case of Markovian transitions one can therefore only compute the *one-step* reachability set for the given (tangible) source states.

Incorporating this functionality into the existing reachability algorithms yields two new variants for symbolic reachability analysis. The pseudo-code of the resulting algorithms is specified as Algo. C.3.A and C.3.B (p. 166). For separately exploring immediate and Markovian activities under a standard bfs-scheme, one generates two different sets of potential transitions. The symbolic structure Z_P holds the set of potential immediate activities and the symbolic structure Z_M the potential transitions as induced by the Markovian ones. Besides the structures holding the unexplored and reached states (Z_U and Z_R), one also maintains the set of vanishing states, as well as the set of tangible states, represented by the symbolic structures Z_{Van} and Z_{Tan}. The SG exploration for models with immediate

activities for a standard bfs-scheme is carried out in the DO-UNTIL loop of lines 5-20 in Algo. C.3.A (p. 166). In line 6 one generates all immediate transitions emanating from the states represented by Z_U. After updating the set of vanishing states (line 9) and restricting the set of unexplored states to the case of tangible ones (line 10), one generates all Markovian transitions emanating from the states represented by symbolic structure Z_U (line 12). This procedure is repeated, until no unexplored state is left. –Algorithm C.3.B similar to Algo. 4.3.B, is organized by following an activity-wise partitioning of the execution of the potential transition functions, where each is represented by the symbolic structures $\widetilde{\mathsf{Z}}_k$. However, as already elaborated above, this needs to be done for immediate and Markovian transitions separately, where in case of the latter the early update strategy for unexplored states cannot be applied. I.e. in the upper inner FOR-loop one first executes the immediate activities, and does so until no new states are reached (line 5 - 11 Algo. C.3.B). What follows next is the *one-step* execution of all Markovian transitions, which gives possibly a new set of unexplored states, represented by symbolic structure Z_U, where this set serves as new input for the next round of SG exploration, starting with the immediate activities. This alternated execution of immediate and Markovian activities is carried out until no new states are reached.

Re-initialization Scheme
It is essential to note, that the set of tangible states as represented by Z_{Tan} may not be correct, it can contain states wrongly assumed to be tangible. Since such states may trigger new immediate model behavior, suppressing any Markovian activity being enabled in such a state. In case of symbolic reachability analysis, this is not problematic, since we are dealing with finite sets of symbolically encoded transition rules, so that the reachability routines will terminate. However, in the context of routine `InitNewRound` new immediate model behavior must be detected and Markovian activities must be prevented from being executed in vanishing states. For doing this one first tests all states if they trigger new immediate model behavior (line 1 - 10, Algo. C.4, p. 166). In this context line 8 also means that there exists no other immediate activity, which can suppress the enabledness of the current immediate activity handled. After processing all immediate activities, the test of new Markovian model behavior is restricted to the remaining states. I.e. in line 11-22 those states which are now (correctly) known to be tangible are tested for triggering new (Markovian) model behavior.

Calculate execution probabilities
In case immediate activities are handled all together, execution probabilities can be obtained, when establishing the transitions as induced by them. But as already mentioned above, in case immediate activities have the same static priority level, it is possible to explore them independently from each other. In such a case the established immediate transitions are equipped with weights, rather than execution probabilities, where the respective activity-specific weight returning function is employed. Consequently one need to transform these weights into probabilities and once a symbolic representation of the overall SG is constructed. This can be achieved by employing a symbolic algorithm, where the weights of transitions emanating from a tangible state are simply normed to their sum.

Eliminating vanishing states
Once the SG is constructed and once the activity weights are converted into probabilities, vanishing states can be eliminated, so that one ends up with a proper CTMC. In [Sie01] algorithms for eliminating vanishing states on the basis of symbolic data types are discussed, this approach can also be employed in the context of this work.

4.7 Related work and own contributions

Published symbolic approaches range from the generation and symbolic encoding of each state individually up to fully symbolic approaches, where explicit SG generation is completely

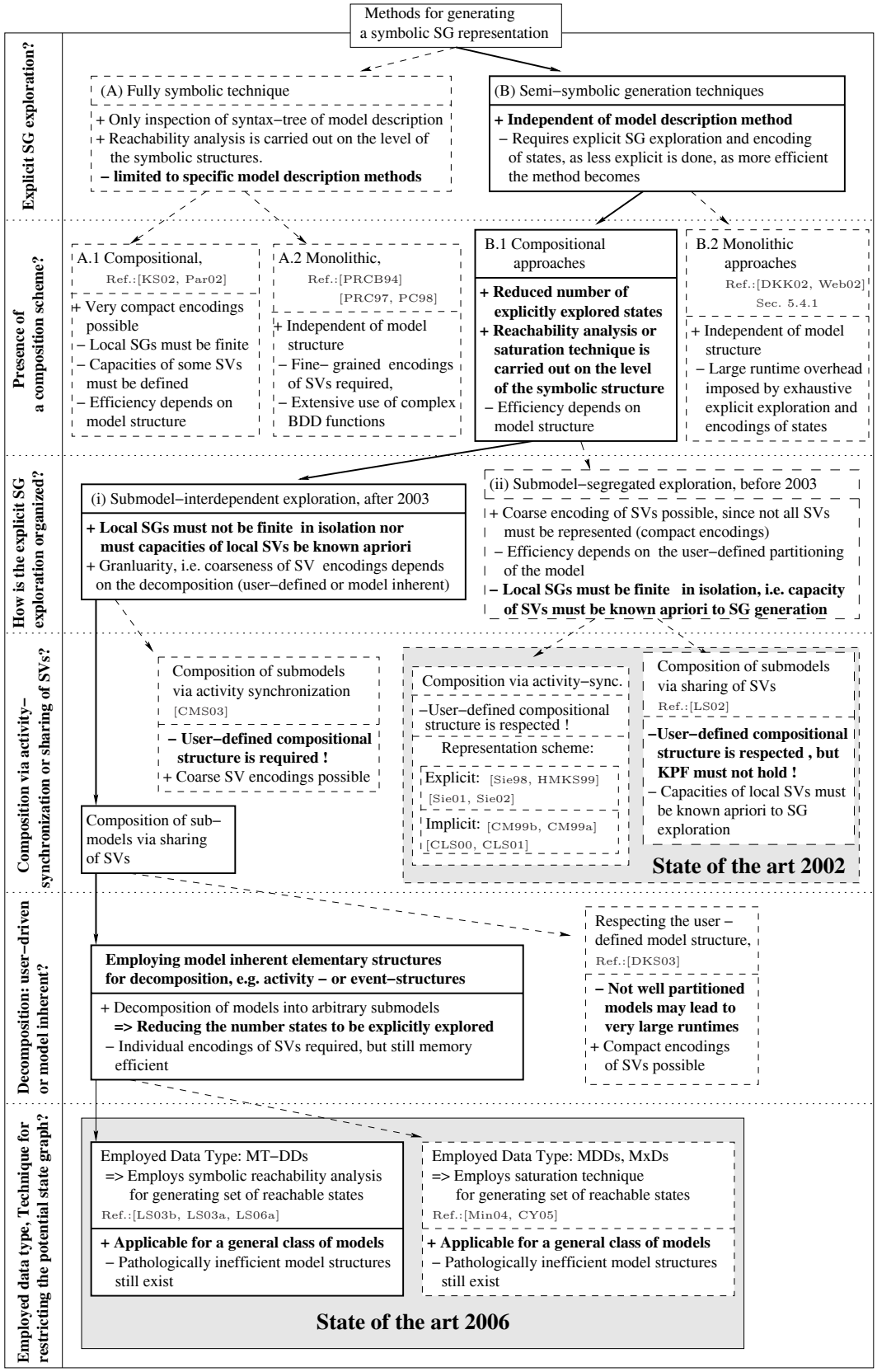


Figure 4.6: Classification of symbolic SG generation methods

avoided. Fig. 4.6 (p. 109) shows a classification of symbolic SG representation schemes. The criteria which drove the process of refining the individual classes of symbolic SG generation schemes are given on the left side. The major line of development, which brought the activity/reward-local approach about is framed by boxes with solid lines. The state-of-the-art of the year 2002, when this research project was started and the-state-of-the-art of the year 2006, –the year a full conference version of the activity/reward-local scheme appeared and this thesis was submitted,– is given in grey-shaded boxes. The pros and cons which drove the design process leading to the activity/reward-local scheme are given in bold faces. In the following the classification will be discussed in greater detail.

At the top level, one may distinguish between fully symbolic and semi-symbolic techniques. Fully symbolic methods require a symbolic realization of the next-state function, which is directly derived from the high-level model description. In contrast, semi-symbolic techniques are characterized by the fact that the symbolic representation of a model's SG is obtained by a combination of explicit exploration and symbolic encoding. At the next level one may split these main categories into the classes of monolithic and compositional approaches. In contrast to the compositional ones, the monolithic approaches do not apply any composition scheme. I.e. they do not take any advantage of a high-level model's structure, let it be user-defined or model inherent. Under a compositional SG generation strategy the high-level model is considered as being decomposable into several submodels.[7] Rather than executing or deriving the symbolic transition function of the high-level model's activities in a general context, this is done on the level of submodels, yielding a set of symbolically represented local SGs. A symbolic representation of the overall SG is then obtained by applying a symbolic composition scheme, which implements the schemes as illustrated in Sec. 2.5 (p. 25). Previous compositional schemes handled the local SGs in isolation, requiring the finiteness of all local SGs. To do so published SG generation schemes often take for granted that either capacities of SVs are computable in advance, e.g. by P-invariant analysis, or that bounds of some SVs are modeled explicitly. Given that the decomposition of flat models into sub-units with finite local SGs of adequate sizes is still an open question, and given that the bounds of SVs is semi-decidable only, the efficient applicability of these methods is clearly limited. From 2003 on, several authors proposed therefore *adapted* semi-symbolic compositional schemes. The newly proposed schemes carry out explicit SG generation in a *submodel-interdependent* fashion, so that neither apriori knowledge about capacities of SVs nor the finiteness of local SGs in isolation is required. This allows one to split the class of semi-symbolic compositional schemes into the class of *submodel-interdependent* and *submodel-segregated* exploration schemes, as done at level 3 of Fig. 4.6. Now the use of different composition schemes allows one to refine these sub-class even further, namely into the schemes applying activity-synchronization and into schemes joining shared SVs.

Besides the applied composition method, one is enabled to refine the subclass of *semi-symbolic compositional submodel-interdependent symbolic SG generation* even further. One can distinguish between methods employing model-inherent structures and methods using the user-defined compositional structure for decomposing a high-level model. The methods of the former class can then be split into the subclass of methods employing *Mt*-DDs and symbolic reachability analysis for restricting the set of states to the actual reachable ones [LS06a, LS06b] and into the subclass of methods employing MDDs or MxDs and making use of the saturation technique for achieving this [Min04, CY05] (cf. level 6 of Fig. 4.6).

What follows next after this general overview is a profound discussion on the individual classes of methods, their advantages and disadvantages, as well as a detailed introduction to the related schemes as found in the literature. However, due to the importance of the semi-symbolic SG generation schemes and in order to elaborate the differences among the

[7] In terms of PNs one may think of a partitioning the overall net into disjoint subnets, where activities are split accordingly (cf. *Sync* driven decomposition as illustrated in Sec. 2.5.3, p. 28) In case of PAs the SGs of the individual processes give the basic partitions a high-level models SG consists of.

recently proposed schemes the discussion of methods of this subclass is devoted to its own subsection.

4.7.1 Fully symbolic techniques

The methods listed here require a symbolic realization of the next-state function, which is directly derived from the high-level model description. Once the symbolic transition rules are generated, symbolic reachability analysis is carried out in order to obtain the set of reachable states. Thus fully symbolic methods are highly efficient, since they avoid any explicit SG exploration, but are limited to some model description methods, e.g. k-bounded non-stochastic Petri nets [PRCB94, PRC97], a restricted input language of the Tipp-tool as employed in the tool Caspa [KS02, KSW04], or the input language of the tool Prism [Pri]. The class of fully symbolic techniques can be divided into the class of models employing a symbolic composition scheme and into the class of methods which treat the model as a flat non-structured unit.

Monolithic techniques

The method presented in [PRCB94] gives an algorithm for generating the set of reachable state for a *non-stochastic, k-bounded* PN on the basis of BDD based state representations.[8] Due to its complexity we limit we discussion to the case of 1-bounded PNs.

Basically each state of the PN is represented by the number of tokens each place currently holds, here at most one. Let the set of states to be explored be represented by a BDD B, i.e. at the beginning B encodes the initial state of the net. Let the set of activity l's dependent SVs be encoded by the sets $\bullet\mathcal{V}_j$ and $\mathcal{V}_j\bullet$, so that the variables of the former set encode l's input places and the latter l's output places.

Each individual exploration step consists of three sub-steps:

(1) Extracting the enabling states for an activity j: This is achieved by conjunction of B and the symbolic representation of the minterm function: $\mathsf{B}' := \bigwedge_{\bullet\mathcal{V}_j} \mathsf{v}_i$.

(2) Destroying the tokens in the *input* places: What follow next is the re-movement of those boolean variable $\mathsf{v}_i \in \bullet\mathcal{V}_j$ from the set of variables of B', which encode the *input set* of the current activity j. This can be done either by co-factorization, as proposed in [PRCB94], or by abstracting from the respective variables ($\mathsf{B}'' := p\texttt{ZAbstract}(\mathsf{B}', \bullet\mathcal{V}_j, +)$). A subsequent conjunction of B'' and the symbolic representation of the minterm function $\bigwedge_{\bullet\mathcal{V}_j} \neg\mathsf{v}_i$, gives the set of states where the input places of activity j are made empty.

(3) Generate the tokens in the *output* places: One need to remove the variables $\mathsf{v}_k \in \mathcal{V}_j\bullet$, which represent the status of the *output places* of the current activity j from the set of already preprocessed states. Analogously to the above step, this is either achieved by co-factorization or by abstraction. A subsequent conjunction with the symbolic representation of the minterm function $\bigwedge_{\mathcal{V}_j\bullet} \mathsf{v}_i$ yields the set of states reachable from the states of B in one step by executing activity j.

If the activities are all executed in a single step, this gives one the image of B under $\mathcal{A}ct$. This symbolic image computation can be incorporated into a reachability algorithm. This reachability algorithm is repeated until a fixed point is reached and the set of all reachable states is generated. The above illustrated approach can be extended to the case of k-bounded PNs, as already done in the original work [PRCB94]. In a later work this approach was extended to a more general class of PNs by introducing a symbolic semantics for inhibitor arcs [PC98]. Since a PN with two inhibitor arcs has "Turing-power" [Hac76], this approach can be in principle employed for any model description method, as long as an efficient

[8] For generating a symbolic representation of the transition relation, this approach needs to be extended.

mapping to a PN exists.[9] For complex model description methods this monolithic and pure-symbolic approach seems to be cumbersome. Furthermore the peak number of BDD-nodes may also be a problem, since it is known that for (semi-symbolic) monolithic methods this number tends to explode, so that these approaches are not applicable in practice (cf. Sec. 5.4.1, p. 137ff).

Compositional techniques

In [Par02] and [KS02] two fully symbolic approaches in the context of stochastic process algebras were presented.

Caspa

The authors of [KS02] achieve a generation of the potential transition system of the overall model as follows: By generating and iteratively traversing a parse tree until a fixed point is reached, each submodel of a high-level model description is translated into its own symbolic structure. A symbolic representations of the high-level model's potential SG is than obtained by employing activity synchronization on the level of the symbolic structures (cf. Sec. 2.5.3 (p. 27ff).

Prism

In contrast the approach implemented in the tool Prism [Pri] employs BDD based algorithms, which directly implement the operator of the input language on the level of symbolic SG representation. Each module of the overall model specification is then symbolically explored until a local fixed-point is reached. A symbolic representation of the high-level model's potential SG is then also obtained by employing activity synchronization, i.e. a realization of the *KO* driven composition scheme illustrated in Sec. 2.5.3 (p. 27ff). However since Prism [Pri] also employs BDDs and ADDs, the handling of global variables within a compositionally constructed high-level models is not problematic.

Irrespective the employed method, both tools carry out the standard bfs based symbolic reachability analysis, so that unreachable transitions can be removed from the set of transitions.

4.7.2 Semi-symbolic techniques

Semi-symbolic techniques are characterized by the fact that the symbolic representation of a model's SG is obtained by a combination of explicit exploration and symbolic encoding, making this methods independent of the employed model description method. This class can further be divided into methods which (a) require to explicitly visit only a fraction of the overall SG by employing a composition scheme, or (b) monolithic procedures which do not require compositional models but need to explicitly visit all reachable states.

Non-compositional techniques

These approaches employ a symbolic structure, where traditional SG generation schemes employed a hash table. Consequently these methods can also be characterized as pseudo symbolic techniques. I.e. the SG is generated in a step-wise fashion by exploring *all at a time enabled activities in each state explicitly*. Consequently methods of this subclass suffer from tremendous run-times. But besides this, experiments show that also the peak memory consumption induces a non-tolerable overhead (cf. Sec. 5.4.1, p. 137ff)

[9] A non-efficient procedure would simply translate the state graph of a model into a respective PN, but being already confronted with the SG-explosion problem.

(1) In [DKK02] the reachability set of a stochastic Petri net is generated by successively firing the enabled transitions, one at a time. Each detected state vector is encoded as a BDD and inserted via disjunction into the BDD representing the set of states reached so far. Additional memory savings are achieved in [DKK02] by making use of *P-invariants*, which allow to reduce the number of SVs per state descriptor. But their computation imposes limitations concerning the expressiveness of the employed high-level model description method, here stochastic Petri nets. [DKK02] developed a BDD-package for representing the set of generate states. This package does not incorporate the isomorphism and *dnc*-reduction rule directly into the node allocating function. In order to avoid extensive number of calls to the BDD reduction routines, the authors of [DKK02] only execute them, if a certain memory size is reached. However, the pay-off for this strategy remains unclear, since when employing unique BDD-nodes their administration cause in general only little overhead (hash-function + collision resolution if necessary), but totally avoids the use of computational expensive reduction routines.

(2) In [Web02] a similar approach to the one discussed above was realized. However, in contrast to [DKK02] the author did not make use of *P-invariants* and employed MDDs for representing the set of reachable states.

(3) In order to investigate the disadvantages of such a monolithic semi-symbolic strategy we also implemented such a method for generating a high-level model's underlying state graph.[10] The collected runtime data will be presented in Sec. 5.4.1 (p. 137ff).

Compositional techniques (submodel-segregated)

Given a high-level model, it might be possible to decompose it into a set of submodels, where the individual local SGs can be generated in a conventional, explicit manner. One may now distinguish between semi-symbolic compositional approaches, which generate local SGs in isolation (*submodel-segregated* exploration scheme), or approaches which follow a submodel-interdependent explicit exploration strategy. In order to point out the evolution and differences among the different methods, the discussion will focus for the time being on the *submodel-segregated* schemes as well as on the composition method. Due to their importance, the more sophisticated submodel-interdependent methods will be discussed separately in the next section (Sec. 4.7.3).

With the developed *submodel-segregated* semi-symbolic compositional symbolic SG generation methods the decomposition of a high-level model is driven by the user-defined compositional structure of the overall model. But nevertheless one ends up with a set of symbolic representations, one for each of the submodel's local SGs. What follows next, is the application of a symbolic composition scheme. On the level of a high-level model description method, submodels are composed by either *jointly executing sets of activities* (synchronization) or *sharing sets of SVs* (join). Realizing such schemes on the level of local SGs requires then respective operators:

(1) Composition of models via activity-synchronization: On the level of local SGs this is achieved by applying a realization of the *KO* based scheme illustrated in Sec. 2.5.3 (p. 27ff). Since in such settings a *KO*-compliant structure of the overall model is always present, –the submodels do not share any of their SVs,– also a BDD based composition scheme realizes a *KO* based composition (cf. Sec. 3.5.2, p. 62ff). Such a scheme can be organized as follows:

(1.a) Delayed: A set of Kronecker operations is executed each time an element of the overall generator matrix is needed [CM99b, CM99a, CLS00, CLS01]. Since the entries

[10] The implementation of such a scheme came almost for free, since the existing implementation of the activity/reward-local scheme could be easily modified to the less complex case of a monolithic generation procedure.

of the overall transition rate matrix are computed when need, rather than kept in memory, one speaks here also of implicit representation methods. Under such implicit representation methods one must also summarize the MxD based methods [Min01], where each traversion realizes a set of Kronecker operations.

(1.b) Promptly: One applies a symbolic version of a *KO* based composition scheme directly on the symbolic represented local SGs and keeps the symbolic represented transition rate matrix of the overall model in memory [Sie98, HMKS99, Sie01, Sie02].
Since this strategy stores the transition matrix directly, where each element can be extracted by BDD-traversion, one may speak of explicit representation.

Since both variants make use of a *KO* it is clear, that they are in principle equivalent.

(2) For complex models it might not be possible in general to derive an ordering on the set of SVs, so that the SVs of each submodel appear in a non-interrupted sequence (cf. Sec. 2.5.6, p. 29f). As proven in Sec. 3.5.2 (p. 62ff), a *KO* based composition scheme is not applicable, since the Kronecker product will deliver wrong results. Thus the delayed (symbolic) methods mentioned above can not be applied at all. Here [LS02] develops a symbolic *Join*-operator, which is based on the `Apply`-algorithm or variants thereof and gives a symbolic semantics for composing submodels via SV sharing (cf. Sec. 2.5.4, p. 28). As discussed in Sec. 2.5.6 this scheme is still applicable, where *KO*-driven schemes may fail. However, in contrast to the latter it requires an encoding scheme where the SVs shared among submodels are encoded individually (fine-grained), rather replacing all SVs of a submodel by a local state counter (coarse encodings) as it is possible under activity-synchronization. But one may note that such a coarse encoding may also lead to larger symbolic structures.
Similar to all other symbolic approaches discussed at that time [LS02] advocates the usage of the user-defined submodels, which may lead to inefficiencies if the sizes of the local SGs are not well balanced. Furthermore a submodel-segregated explicit SG generation scheme was assumed, so that *apriori knowledge* of bounds of SVs must be available, in order to explore the local SGs in isolation.

4.7.3 Semi-symbolic, compositional and submodel-interdependent techniques

By the end of 2002 different symbolic SG generation schemes had been developed, realizing either a synchronization *Sync* or *Join* on the level of submodels, but leaving two main problems open:

(1) In general bounds of SVs are not decidable, thus apriori knowledge about them not available, consequently the generation of local SGs in an isolated or submodel-segregated fashion is in practice often not feasible.

(2) In order to decompose a high-level model the approaches discussed so far employ the user-defined compositional structure of the overall model, which may lead to local SGs of unbalanced size. Given that the partitioning of flat models into (independent) submodels with finite local SGs of adequate sizes is still an open question the applicability of the suggested approaches is obviously limited. This situation is even worsen, since in contrast to the approaches making use of the `Apply`-algorithm, like [Sie02, LS02] all other approaches are dependent on a *KO compliant* structure of the high-level model.

In order to overcome this restriction in 2003 different authors suggested schemes, where the local SGs were generated in a submodel-interdependent fashion [LS03b, LS03a, CMS03, DKS03]. As a consequence bounds of SVs do not need to be known apriori nor do the local SGs need to be finite in isolation. However, the problem of a(n) (automatized) partitioning driven by model-inherent structures, rather than specified by the modeler, was solely tackled in [LS03b, LS03a].

[CMS03] employed activity synchronization as method of composition, where the user is

required to decompose an overall model into partitions. Thus the efficient applicability of the presented method is restricted to models, where a *KO compliant* partitioning is possible and the individual partitions yield SGs of similar sizes. On the other hand, the partitioning of the overall model allows the authors of [CMS03] to employ a very dense encoding scheme, since on the level of submodels SVs can be replaced by local state counters (cf. to the data presented in Sec. 5.4.2, p. 140ff).

In contrast [DKS03] suggested a scheme for SV-sharing models and consequently for models where the *KO compliant* structure may not be given. The suggested method employs the user-defined compositional structure of the high-level model, which may yield to local SGs of unbalanced sizes and / or models where a lot of inter-action among the submodels takes place. Thus in practice this approach often yields non-acceptable run-times, when generating a MxD based representation of the overall model transition system, which of course depends on the employed high-level models (cf. to the data presented in Sec. 5.4.2, p. 138ff).

[LS03b, LS03a] presented an approach, where the problem of unknown bounds of SVs or the non-finiteness of local SGs in isolation was also solved by applying a submodel-interdependent SG generation scheme. But in contrast to [CMS03, DKS03], the problem of partitioning the high-level model was also taken into account by maintaining compositionality at the level of individual activities. As a consequence the activity/reward-local scheme, presented in [LS03b, LS03a] did not require any particular compositional structure of the high-level model, nor must the high-level model be specified under a specific description method.[11] Until now others also presented approaches, which are capable of dealing with models, where a *KO compliant* partitioning is not mandatory:

[Min04] suggests a partitioning of the overall model into submodels, but in contrast to the activity/reward-local scheme each SV is a submodel, where the connecting activities are split among the submodels. Each of the resulting event-local structures, which can not be explored separately, since non-resolvable dependency with other event-local structures exists, are merged to form a single submodel. Non-resolvable dependency exists if the firing of an event not solely depends on the submodel-local SVs. As drawback, this strategy may lead to a partitioning, which does not have a *KO compliant* structure. Now and in contrast to his earlier works, the author of [Min04] applies an encoding scheme, where each SV gives its own MxD-level. This encoding allows the application of the same composition scheme as in case of the activity/reward-approach, but in the context of MxDs and by making use of an MxD based variant of the `Apply`-algorithm. The restriction of the potential SG to the set of reachable transitions is achieved by employing this composition scheme within the well known saturation algorithm.[CLS01].

The authors of [CY05] also exploit event-locality, so that in contrast to their earlier works each SV corresponds to its own MDD level. This allows an arbitrary decomposition of the high-level model, where SVs belonging to the same event are combined via conjunction. The disjunctive composition of the event inducing PN transitions gives one a symbolic representation of the potential SG. In this context it is interesting to note that in [CY05] the same notion of independence of SVs and activities/events as already employed in [LS03b, LS03a], is used. Furthermore the authors suggest also the above mentioned partitioning of the set of activity-dependent SVs into enabling and updating parts. The different symbolic structures encoding the enabling and updating function of an activity for a given pair of states as contained within the transition relation of a model are then combined via conjunction. This is exactly what the explicitly encoding of the individual transitions does in case of the activity/reward-local scheme, since each symbolically encoded transition gives one a minterm of the respective boolean variables, let them be partitioned into sets of enabling

[11] The approach of [LS03b, LS03a] is then only applicable when independent sequences of activities do not trigger new model behavior. However, this problem was easily resolved by introducing a re-initialization routine, so that more rounds of submodel-interdependent SG generation and transition encoding may follow, if necessary.

and updated SVs or not. Finally the disjunction over all event-induced transitions is clearly analogous to the summation in case of the activity/reward-local scheme. However, since the authors of [CY05] introduce an *identity-reduction* rule for MDDs, the explicit insertion of identity structures as in case of the activity/reward-local scheme is unnecessary. As a result of the disjunctive and conjunctive partitioning and the subsequent composition in relation with the guarantee of identity-free MDDs, the authors of [CY05] are finally enabled to present a saturation algorithm, which efficiently handles models with unknown bounds of SVs and does not require a (*KO* conforming) user-defined compositional structure as it was the case in [CMS03].

The above illustrated methods are based on summation of potential submodel-local transition functions in order to obtain the potential transition function of the overall model. Each potential submodel-local transition rate function is hereby obtained by cross-product building of the real submodel-local transition matrix with adequate identity structures, where in case of identity-free MDDs this step can be omitted. Since the summation as well as the cross-product building is realized with algorithms, which are in fact variants of the `Apply`-algorithm, called with the respective `op`-function, it is clear that this composition scheme employed in [Min04] and [CY05] are equivalent to the composition scheme of the *symbolic join* as presented in [LS02] and to its more generalized version as employed in the activity/reward-local scheme [LS03b, LS03a]. Besides the employed data structures and the chosen partitioning of the model another difference of the activity/reward-local scheme and the work presented in [Min04] and [CY05] can be revealed: In [LS06b] the local character of activities is extended to the case of rate - and impulse reward functions, yielding highly efficient schemes for generating a symbolic representation of user-defined PVs. Since their symbolic representation not only allows a memory efficient storage, but also an efficient computation of their moments, the efficient calculation of performance variables is achieved, where [Min04] and [CY05] are only concerned with symbolic SG generation.

4.7.4 Symbolic algorithms for generating the set of reachable states

The fully symbolic and monolithic approaches organize their reachability analysis on the level of symbolic structures. This also holds for all compositional schemes, since in contrast to the (semi-symbolic) non-compositional schemes they generate the potential transition system and therefore require a scheme for generating the set of actually reachable states. Besides the different layouts of the various (compositional) schemes discussed so far, another important difference can be revealed. This difference is related to the employed data types. Approaches making use of BDDs or variants thereof relay on standard bfs symbolic reachability analysis, whereas MDD or MxD based approaches make use of a technique called saturation, which was first introduced in [CMS03]. This technique directly manipulates the node pointers between adjacent MDD, MxD-levels, when exploring local model behavior, which may lead to run-time advantages for some models. Since for the BDD based compositional schemes symbolic reachability analysis is the main source of CPU-time consumption, it is clear that from an improvement here, the overall scheme will benefit. In [LS06a] a new scheme for symbolic reachability analysis was presented. This scheme is based on the idea of executing the symbolic transition functions in an activity-wise manner, rather than executing them all at once. Thus similar to the approach of [BCL91] one executes partitions of the overall transition system sequentially. It seems that the handling of smaller DD-structures is more efficient than handling huge DD structures once at a time. Furthermore an activity-wise refinement enables one to employ an *early-update strategy* on the set of states to be explored in the next step. As we recently noticed in the technical report [PRC97] also an activity-wise instead of a *"all-at-once"* image computation is advocated. Doing so enables the authors of [PRC97] to use also all newly reached states when symbolically executing the next activity. This technique, which we denoted as early-update strategy, was denoted by the author of [PRC97] as *greedy chaining*, in reminiscence of the greedy heuristic for algorithms. Thus the proposed scheme in fact turns out to be very similar to our new symbolic reachability scheme as introduced in Sec. 4.3.4. However, [PRC97] is limited to the

case of k-bounded non-stochastic Petri nets having a symbolic semantics for the individual firing rules of the activities and does not employ a semi-symbolic compositional scheme. Furthermore, it is interesting to note that the symbolic reachability analysis as carried out there relies on the extensive use of BDD-operations. In contrast to this monolithic BDD based approach, the compositional BDD based approaches, let them be semi- or fully symbolic, generate a symbolic representation of the potential transition system. Consequently the symbolic next-state function is encoded within a **single** symbolic structure, or at least a symbolic structure for each activity, simplifying the process of image computation to a single conjunction, abstraction and re-labeling operation. Consequently this simplification should lead to runtime advantages of these methods over the non-compositional and fully symbolic approaches of [PRCB94, PC98], as far as the symbolic reachability analysis is concerned.

4.8 Pre-published material

[LS02] describes how to construct complex performability models in the context of the software tool Möbius by hierarchically composing small submodels. In addition to Möbius' "Join"-operator, a second composition operator "Sync" is introduced, and it is shown how both types of composition can be realized on the basis of symbolic, i.e. BDD based data structures. I.e. in case of the "Join"-operator a respective symbolic composition scheme is developed, so that one is enabled to combine submodels via matching of the values of the SVs shared among the submodels. This composition scheme is a prerequisite, when it comes to the activity/reward-local scheme. Thanks to Bryant's `Apply`-algorithm and its variants, the composed submodels do not need to have a *KO compliant* structure. However, in [LS02] it was assumed that the local SGs can be generated in isolation. This assumption turned out to be not realistic, due to the lack of the apriori knowledge of the bounds of the shared SVs. In order to solve this problem we designed a scheme, which iteratively and in an interleaved fashion explicitly explores as few activities as possible. The basic ideas of this scheme, which we denoted as activity/reward-local SG generation and representation scheme, were presented in [LS03b, LS03a], but for a limited class of models and with a standard symbolic reachability analysis. In [LS06a] the basic scheme was extended, so that one is now capable of handling models, where the combined execution of independent activities triggers new model behavior. Furthermore [LS06a] introduces the new "quasi"-dfs symbolic reachability analysis, where partitions of the transition function are executed sequentially rather than all at once. The generation and handling of reward functions in order to compute complex performance measures on the basis of a symbolic SG representation was than contributed in [LS06b].

5

Empirical Evaluation

In this chapter we will assess the major contributions of this thesis by analyzing models which are well-known from the literature and often employed as benchmarks. For providing evidence of the usefulness of the newly introduced type of DD (*p*ZDDs) and the activity/reward-local scheme, this chapter compares also the run-time data of our implementation to the run-data of other well-known performability tools. For demonstrating the practical applicability of the developed contributions, we also present the analysis of a case study. In this study the availability of a telecommunication service system had to be computed, which required the solution of a MRM consisting of more than 260 million system states.

5.1 Organization of the chapter

Sec. 5.2 gives details about the implementation and the models used as benchmarks. Furthermore it introduces the different settings, as employed for investigating the performance of the (p)ZDD based framework for the SG based performability analysis.[1] In Sec. 5.3 the efficiency of the the activity/reward-local SG generation technique is investigated, where different aspects are considered. At first we compare the performance of ZDDs and ADDs, if employed within the activity-local scheme. What follows next, is a comparison of the standard bfs and our new quasi-dfs symbolic reachability algorithm. The assessment is made complete by investigating the significance of the variable ordering for the efficiency of the activity/reward-local scheme and its symbolic algorithms.
Sec. 5.4 presents a comparison of our ZDD based activity/reward-local scheme to other symbolic SG generation techniques. These techniques range from fully symbolic ones as incorporated into the tools Caspa [KSW04] and Prism [Pri] over semi-symbolic monolithical procedures, up to the semi-symbolic submodel interdependent approaches developed recently and currently incorporated into the tools Möbius [DCC+02] and Smart [Sma].

Sec. 5.5 presents an assessment of the ZDD based hybrid solution method for computing state probabilities of a MRM. At first a comparison of ADD- and ZDD based hybrid solvers is carried out. As next the impact of the choice of sparse and block-level on the ZDD based solvers is investigated. The assessment of the solvers is made complete by investigating the significance of the variable ordering on the performance of the solvers.
Sec. 5.6 compares the ZDD based solvers to other solvers as realized in the tools Möbius and Smart. Their memory management for storing matrices range from traditional sparse matrix layouts over Kronecker operator based sparse matrix representations up to symbolic formats such as multi-valued multi-terminal decision diagrams (MTMDDs) and the well-known matrix diagrams (MxDs) of [Min01].

In Sec. 5.7, a case-study of a telecommunication service system will be presented. This case-study is also fed into the *DSPNexpress* tool, so that another comparison of our ZDD based activity/reward-local scheme, and a standard performance evaluation tool is achieved.

Sec. 5.8 will conclude this chapter by indicating our pre-published material.

[1] For the time being we simply ignore the fact that the symbolic structures may partially shared and speak generically of ZDDs.

5.2 Preliminaries

The algorithms discussed so far constitute a symbolic framework for carrying out the SG based analysis of high-level MRMs. This framework is capable of either employing ADDs or ZDDs when generating and analyzing the low-level MRM. Different aspects of an implementation of this framework within the Möbius modeling tool will be presented now.

Möbius [DCC+02] is highly suited for implementing the activity/reward-local approach for the following reasons:

(1) The Möbius modeling framework supports several model specification formalisms to be combined within a single hierarchic overall model. Therefore the application of a purely symbolic SG generation method seems disadvantageous, since each newly implemented model-description method would require a new symbolic semantics. Consequently the employment of a semi-symbolic technique is appropriate.

(2) Due to the nature of the *Join* and *Rep**Join* model composition method and the fact that one cannot calculate the capacities of SVs in advance, the Möbius modeling framework does not support a compositional SG construction, if the submodel-local SGs are generated in isolation. As a consequence, one has to employ one of the submodel-interdependent SG generation techniques, since they are capable of handling such situations.

(3) Within the Möbius modeling framework the modeler builds a (hierarchic) high-level model from an arbitrary number of submodels, where a submodel is either a hierarchic model itself or a basic model description, commonly denoted as atomic model. In order to obtain a hierarchic model, the atomic models are combined via the *Join*, *Rep**Join* or *Sync* composition operator. The sizes of the local SGs of the atomic models are not known beforehand. Thus trying to exploit the user-defined compositional model-structure for constructing the overall SG from the local SGs of the atomic models may not be efficient at all. In such cases the activity-local method offers the advantage of being independent of the user-defined compositional model-structures, since it employs model-inherent structures.

The implementation of our symbolic framework as incorporated into the Möbius modeling tool consists of four modules:

(1) A module for the explicit generation of transitions, which constitutes the interface between the symbolic engine and Möbius (`ExploreStates`, Algo. 4.2.A, p. 88).

(2) The symbolic SG generation engine, providing the following features:
- Symbolic encoding of transitions and generation of the set of activities to be explicitly executed (`EncodeTransitions`, Algo. 4.2.B, p. 88),
- the symbolic composition scheme for generating the set of potential transitions (cf. Sec. 4.3.3, p. 89f),
- the symbolic reachability analysis, in order to obtain the set of reachable states and transitions (`ReachabilityAnalysis`, Algo. 4.3, p. 90), and
- the re-initialization scheme, triggering new rounds of the activity/reward-local scheme (`InitNewRound`, Algo. 4.4, p. 91).

In combination the above features deliver an *Mt*-DD based representation of the high-level model's underlying (activity/reward-labeled) CTMC.

(3) A *Mt*-DD-library based on the CUDD package [Som98], providing our implementation of *p*ZDDs and CUDD's originary implementation of ADDs. I.e. in case one makes use of ZDDs, the library provides the C++-class definition of *partially shared ZDDs*, the new recursive algorithms for manipulating them, the `op`-functions and an implementation of the operator-caches, which are essential for efficiently manipulating *p*ZDDs. In case of

N	$states$	$trans$	$trans_e$
FTMP			
2	$2.5693E5$	$1.6978E6$	688
3	$1.2408E8$	$1.1513E9$	1.548
4	$5.5039E10$	$6.6113E11$	2.752
5	$2.3549E13$	$3.4847E14$	4.300
6	$9.9082E15$	$1.7463E17$	6.192
Courier			
4	$9.7102E6$	$5.7005E7$	142
5	$3.2405E7$	$1.9983E8$	206
6	$9.3302E7$	$5.9818E8$	289
7	$2.3965E8$	$1.5858E9$	394
8	$5.6182E8$	$3.8166E9$	524
TQN			
63	$8.128E3$	$2.7971E4$	$8.127E3$
64	$8.385E3$	$2.8863E4$	$8.384E3$
127	$3.2640E4$	$1.1328E5$	$3.2640E4$
128	$3.3153E4$	$1.1507E5$	$3.3153E4$
255	$1.3082E5$	$4.5594E5$	$1.3082E5$
256	$1.3184E5$	$4.5952E5$	$1.3184E5$
511	$5.2378E5$	$1.8294E6$	$5.2378E5$
512	$5.2583E5$	$1.8365E6$	$5.2583E5$
1023	$2.0961E6$	$7.3288E6$	$2.0961E6$

N	$states$	$trans$	$trans_e$
Kanban			
5	$2.5464E6$	$2.4460E7$	1.860
6	$1.1261E7$	$1.1571E8$	4.116
7	$4.1645E7$	$4.5046E8$	8.232
8	$1.3387E8$	$1.5079E9$	15.192
9	$3.8439E8$	$4.4746E9$	26.280
10	$1.0059E9$	$1.2032E10$	43.120
FMS			
5	$1.52712E5$	$1.1115E6$	290
6	$5.3777E5$	$4.2057E6$	434
7	$1.6394E6$	$1.3553E7$	616
8	$4.4595E6$	$3.8534E7$	840
9	$1.1058E7$	$9.9075E7$	1.110
10	$2.5398E7$	$2.3452E8$	1.430
12	$1.11415E8$	$1.07892E9$	2.236
Polling			
15	$7.3728E5$	$6.144E6$	60
18	$7.0779E6$	$6.9599E7$	72
20	$3.1457E7$	$3.4078E8$	80
21	$6.6060E7$	$7.4868E8$	84
25	$1.2583E9$	$16.7772E9$	100

Table 5.1: Model specific data for the various case studies

ADDs this library employs the original C++-classes, algorithms and caches as provided by the original CUDD package.[2]

(4) A library for computing the user-defined PVs (cf. Sec. 4.5, p. 99ff.), implementing function `ComputePV` (Algo. 4.5,p. 100) and its employed routines, such as:
   (4.a) `ComputeStateProbabilities` providing the new ZDD based hybrid solvers and a version of the ADD based hybrid solvers as used within the tool Caspa and Prism, which enable one to compute steady state or transient state probabilities of the generated CTMCs.
   (4.b) `MakeRateRewards` and `MakeImpulseRewards` (Algo. 4.6, p. 101) which efficiently generate symbolic representations of rate and impulse reward functions, and
   (4.c) `ComputeRew` for computing the first and second moment of each user-defined PV via *Mt*-DD-traversal (Algo. 4.7, p. 103).

5.2.1 Employed models for benchmarking

For evaluating the symbolic framework several models, which are commonly employed as benchmarks in the literature, will be analyzed. Tab. 5.1 gives the sizes of their SGs, i.e. the number of states (*states*) and the number of transitions (*trans*), as well as the number of transitions explicitly explored by the activity/reward-local scheme ($trans_e$). These characteristic dates depend on the respective model scaling parameter N, which is also given. In the following paragraphs, these models are briefly introduced. Issues related to their SG based analysis will be already discussed a long the way, so that the data presented later can be assessed correctly.

[2] Currently the implementation of *p*ZDDs within the JINC DD-packages is in progress. In contrast to the CUDD DD-package [Som98], the JINC package [Oss06] is truly C++-oriented and seems to be more efficient when it comes to the manipulation of DDs.

Kanban Manufacturing System (Kanban) [CT96]

The Kanban model describes a Kanban production system with four cells, where each cell has a single type of workpiece and Kanban cards. Since the manufactured pieces may not match their specification, each cell has the possibility to re-work the pieces and re-insert them into the production cycle. The assembling of parts from the individual workpieces of the cells is modeled by two complex activities. The number of workpieces and Kanban cards per cell and per workplace is fixed and adjusted by the model scaling parameter N, so that the model can be scaled by changing this value. Workpieces and Kanban cards are represented by the tokens circulating in a cell, where the places of each cell are N-bounded. The Kanban model is thus highly suited for being analyzed by making use of the *Sync* driven decomposition of the high-level model, where the overall SG is obtained by applying a *KO* driven composition scheme to the submodel-local SGs (cf. Sec. 2.5.3, p. 28ff). To do so, the afore mentioned complex activities, which represent the assembling of parts, are split into cell-local activities, so that one ends up with four independent submodels, each representing one single production cell. For obtaining the model's overall SG from the submodel's local SGs, activity synchronization over the previously split activities takes place. The partitioning of the overall model into four submodels not only allows to construct the submodel-local SGs in isolation, but also leads to a well balanced partitioning of the overall SG. Thus, not surprisingly, the Kanban model is highly suited and often used for illustrating the good performance of compositional symbolic SG generation methods, which employ the *Sync* composition method and thus a *KO* driven composition scheme for constructing a symbolic representation of an overall model's transition rate matrix.

Flexible Manufacturing System (FMS) [CT93]

The FMS model specifies a manufacturing system consisting of four machines. Hereby two machines process only their own workpiece, the third machine processes either its own workpiece or, if being idle, it processes workpieces of one of the other machines (always of the same machine). The remaining machine assembles a more complex workpiece, where the above three basic workpieces are employed. I.e. in total the FMS produces four different parts, which leave the production process as soon as they are finished. In order to maintain a constant inventory, the number of rough workpieces entering the system is equal to the number of finished parts leaving it. Rough workpieces and finished parts are moved between the machines by employing pallets, where the capacity of the pallets depends on the kind of the workpieces loaded. The total number of workpieces of the basic kind $i \in \{1, 2, 3\}$ is specified by $N_i = N$, where that N is the FMS model's scaling parameter. The total number of pallets within the FMS is defined as $3N$. In the original model, which will be denoted as FMS (orig.), the rate of the activities modeling the feeding of rough workpieces into the respective machine, depends on the number of rough workpieces in their input places, multiplied with $min(1, \text{ratio\_of\_number\_of\_all\_pallets\_and\_available\_ones})$. Since the number of available pallets is given as the sum of tokens over a set of places, this significantly enlarges the number of dependent SVs of the afore mentioned activities. As a simplification we will omit this ratio, so that the rates are solely determined by the number of tokens in the input place of the respective activity. This simplification was also applied in the work of others, e.g. [KSW04]. Due to the complex structure of the FMS model the finding of a *KO compliant* decomposition is not straight forward. It either requires a re-specification of the model or clearly decreases the efficiency of the compositional SG construction methods, which are based on activity synchronization.

Cyclic Polling System (Polling) [IT90]

The Polling model represents a cyclic server-polling system with N stations. A station is either idle, waiting for processing a service request or, if service is granted by the server, processing the latter. The Polling model can be modeled as a hierarchic model, where interaction among stations and client is modeled by the *Sync* or *Join*-method. In case of

activity-synchronization, the overall model is built from a set of client stations and a server station. Service is granted to a client station, if the requesting client and the server are capable of jointly executing a dedicated activity. In case of the *Join*-method, there is no need for modeling the server station explicitly. The granting of service is modeled by a token, which is handed over from client station to client station. Each client passes the token to its neighbor (uni-directionally) as soon as its service request is processed or, in case nothing needs to be done, after an exponentially distributed time delay. In both cases, the Polling model is one-bounded and produces identical SGs underlying the different model specifications. Both forms of composition (*Sync* or *Join*) allow a very efficient construction of the overall SG.

Tandem Queuing System with Blocking (TQN) [HMKS99]

The TQN model consists of two stations. The first is an $M/Cox_2/1/N$ queuing-station which is coupled in a blocking manner with a second $-/M/1/N$ queuing-station. The input queues of both stations have the same limited capacity, modeled by parameter N. The arrival at station 1 is a Poisson stream, where its service is modeled by a Coxian distribution with two exponentially distributed phases. The second service phase is entered with probability $1 - p$, and it is skipped with probability p. After being processed in queuing-station 1, a served job enters the input queue of the second station. The service duration of this station is exponentially distributed. In this work, the TQN model was specified as SAN consisting of 3 places, two places for representing the input queues and one place for storing the job, which is scheduled to enter service phase two. Consequently, two of the places may contain the number of tokens specified by the scaling parameter N, and the remaining place contains either one or zero tokens. In contrast to [HMKS99] station 1 is fully blocked when the input-queue of station 2 is full. One may note that the TQN model constitutes a worst case scenario for the activity/reward-local scheme and also for methods based on the *Join* composition method. This stems from the circumstance that the high-level model consists of 3 SVs only, so that a high degree of dependency among the activities or user-defined submodels exists, yielding a large number of transitions to be explored explicitly. In contrast to this, a *Sync* driven decomposition (cf. Sec. 2.5.3, p. 28) enables one to encapsulate the places into independent submodels, and thus reduce the number of explicit exploration steps clearly, so that the generation of a symbolic SG representation can be achieved more efficiently. A solution of this problem will be sketched at the end of this thesis (cf. Sec. 6.3, p. 159)

Fault Tolerant Multi-Processor System (FTMP) [MS92]

The FTMP model consists of a multi-processor system, built from redundant components. Therefore this model is highly suited for being specified by making use of the *Rep**Join* composition method. In contrast to the original model, we did not take advantage of the fact that components may be identical. Thus we do not take advantage of user-defined symmetries for generating a reduced SG. In our setting the model consists of N individual processor systems, each constructed from the following individual components:

- A Memory module consisting of three memory units, where at least two must be operational. Each memory unit possesses 41 RAM chips, of which at most two may fail, and two interface chips that must all be operational.
- A processor unit containing three CPUs of which two must be operational.
- An I/O port module containing two ports of which one must be operational. In total, CPUs and I/O-ports consist of 6 non-redundant chips each.
- A module for error-handling.

A processor system fails, if one of its components fails. The overall system is operational if at least one processor system is operational.
Since the degree of dependency among the different components, each encapsulated into

its own submodel, is very high, one faces a high number of interactions between the latter when exploring the local SGs in a *submodel-interdependent* style. Thus symbolic approaches exploiting the above illustrated user-defined structure, and a *Join* composition method may not be very efficient.

Courier Protocol (Courier) [WL91]
The Courier model specifies a parallel communication software system . The model's scaling parameter N gives the size of the transport window (TWS), so that the number of packets simultaneously transmitted between sender and receiver is bounded by this size. In this work, we employed the model of [DKS03], where the communication software system is constructed from four submodels: the receiver's session and transport layer, and the sender's session and transport layer. The four submodels are composed via the *Join* composition method. In contrast to the FTMP model, the Courier model seems to exhibit more local behavior. Not surprisingly the symbolic SG generation procedure of [DKS03], which depends on the user-defined partitioning and the *Join* composition method, performs here much better.

5.2.2 Layout of presented run-time data

The tables presented in this chapter will always contain the run-time data as gathered for the different models and for the different experiments. Hereby columns filled with xxx correspond to experiments which could not be carried out by the respective approach, due to memory and/or run-time restrictions. Columns filled with ??? correspond to data, which could not be obtained from the specific experiment and for the specific tool employed.

For simplifying the comparison of the different methods, we decided to present not only the pure run-time data, but to present also ratios, where the respective figures are always normed to the data as produced by our ZDD based implementation. I.e. ratios > 1 normally indicate an advantage of the innovations developed in this work, and ratios < 1 indicate their disadvantage. If this is not the case, it will be stated.

Since the data presented in this chapter was taken at different stages of this project, the run-times may vary. However, these variation are not only due to improvements of our implementation, but also to the employment of different C++ compilers. But these differences are of minor importance, since the innovations developed in this work clearly improve symbolic SG based analysis of high-level MRM by reducing run-time and space complexity by significant factors.

5.2.3 Platform

All benchmarking experiments as carried out with our implementation and with other tools, were executed on Pentium 4 systems with a Linux OS, where we employed either 3 GHz or 2.88 GHz machines. I.e. the figures of Tab. 5.2 - 5.5, Tab. 5.7 and Tab. 5.14 were produced on a P4 with 3.0 GHz and 1 GByte of RAM, whereas the figures of all other tables were produced on a P4 with 2.88 GHz and a maximum of 4 GByte of RAM.

Since the memory space of a 32-bit machine is currently restricted to assigning at most 3 GByte to a single process [Intb], the case study, presented in Sec. 5.7, had to be analyzed on a 64-bit machine. We employed the computing cluster of the Technical University Munich, where we used the AMD Opteron 850 with 2.4 GHz, equipped with either 8 or 16 GByte of RAM and a Linux OS [Inta].

5.2.4 Comparisons

For evaluating the contributions of this work, various comparisons are made.

Symbolic SG generation approaches

For assessing ZDDs and the activity/reward-local scheme comparisons to other symbolic SG generation techniques will be carried out. The considered techniques range from the fully symbolic ones over semi-symbolic monolithical up to the semi-symbolic submodel interdependent techniques developed recently.

Compositional and fully symbolic approaches

The tools Prism [Pri] and Caspa [KSW04] are both based on the CUDD package, which we also use. This is a very important aspect, since when employing the ADD implementation of the JINC package [Oss06] instead of the implementation of the CUDD package, the SG generation times can already be reduced clearly. Thus the run-time reduction stemming from the implementation of the respective data type is a crucial factor, but often remains unclear. This is why we only employed the CUDD package in the following.
Since only Prism, similar to our own implementation, individually encodes the different SVs within the symbolic structures, the focus of the comparison was on Prism rather than Caspa. This is justified, since symbolic reachability analysis is the dominant run-time factor and Caspa generates flatter *Mt*-DD structures, increasing the effectiveness of their manipulation. A comparison with Prism is one of the most meaningful assessments, since our implementation and Prism not only employ the CUDD package, but also the ADDs for representing the set of reachable states and transitions as generated by Prism and our implementation are isomorphic, as long as the model descriptions and the chosen variable orders match. In contrast to Prism, which orders the SVs by the sequences of their occurrence within the model specification, where global variables come first, our implementation of the activity/reward-local scheme allows an arbitrary ordering of the Boolean vectors encoding the SVs. This feature will be important when investigating the influence of the orderings of SVs on the performance of the symbolic SG techniques.

Monolithic semi-symbolic approaches

As already reported in Sec. 4.7.2 (p. 112) an implementation of a monolithic and semi-symbolic SG generation procedure within our ZDD based framework came almost for free. It is highly suited to demonstrate how important compositionality for a SG generation scheme is, since under the monolithic procedure it turned out, that the peak number of ZDD-nodes explodes and the explicit encoding of all transitions induces a non-acceptable run-time overhead. Thus, a comparison of the activity/reward-local scheme and a monolithic semi-symbolic approach will be presented.

Compositional, semi-symbolic and model-interdependent approaches

We compared our implementation of the activity/reward-local scheme to the MDD based approach of [DKS03], because our implementation is within Möbius and - similar to [DKS03] - uses the Möbius high-level model specification and the standard next-state function of Möbius for explicit exploration. For the sake of completeness we also compared our implementation to the tool Smart. The release of Smart, which we obtained in May 2006, only contains symbolic SG-generation methods which were published before 2004 (cf. [CM99b, CM99a, CLS00, CLS01, CMS03]). A comparison to the tool Smart needs to be considered with care: (a) Since the methods described in [Min04, CY05] are not part of the release of Smart, which we obtained in May 2006, the comparison is restricted to those methods, which employ user-defined partitions for generating a symbolic SG representation. I.e. in contrast to our implementation, the methods implemented in Smart rely on a *Sync* driven decomposition of the overall high-level model, so that a symbolic representation of the high-level model's SG can be obtained by applying a *KO* driven composition scheme (cf. Sec. 2.5.3, p. 28). Therefore the presented run-time data of Smart is highly dependent on the user-defined partitioning of the overall model, which we will demonstrate by investigating differently partitioned Kanban models. (b) Furthermore, one needs to take into consideration

that there is neither a common implementation framework, nor a common data structure between our Möbius implementation and the tool Smart, making the comparison of run-time data for evaluating the symbolic SG generation techniques inaccurate (see discussion concerning JINC and CUDD from above).

Comparison of solvers

The performance of the solvers for computing steady state and transient state probabilities is one of the major issues, since in contrast to symbolic SG generation schemes, their computation may take several hours or even days. For assessing our implementation of the hybrid ZDD based solvers we compared them to: (a) the hybrid ADD based solvers as included in the tools Prism and Caspa, (b) the standard sparse matrix solvers of the Möbius tool, and (c) the non-symbolic and symbolic solvers of the Smart tool. This assessment will round out the benchmarking of the implemented symbolic framework.

5.3 Assessing the activity-local SG generation scheme

5.3.1 Comparing ADD and ZDD based SG generators

Tab. 5.2 illustrates the different run-time data as measured when generating the SG of the different benchmark models by employing the activity-local scheme in combination with either ZDDs or ADDs. The first column gives the model scaling parameter (N), the second gives the total number of Boolean variables employed for encoding all SVs, each state respectively ($n_{\mathcal{V}}$). Each SV was hereby encoded by a minimum number of bits, where in practice such an allocation strategy is not feasible for ADDs, due to the lack of a priori knowledge of the maximum value K_i taken by SV s_i. This means that practical figures would be even more favorable for ZDDs, since under a brute-force strategy, where one allocates as many ADD variables as possible, a significant increase in memory space and run-time for the ADD based SG generator would be observable. In contrast to ADDs, pre-allocation of Boolean variables for ZDDs is almost effortless. Skipped variables are interpreted as being 0-assigned and thus one only allocates here a node, if the associated bit-position is really needed (cf. Sec. 3.5.1, p. 58).

Tab. 5.2.A and B give the number of nodes required for representing (a) the set of reachable states (encoded by Z_R or A_R), (b) the transition system (encoded by Z_T or A_T), and (c) the peak number of nodes allocated during the process of symbolic SG construction (*peak*). In our implementation, since we employed the CUDD-package, each node consumes 16 bytes of memory, where in principle 10 Byte would suffice (2 pointers and one level index counter). The columns t_g contain the SG generation time in seconds. Furthermore we provide the ratio of the hit rates of the op-cache (r_{chr}), since it is also a good indicator for the efficiency of the employed type of DD.

In Tab. 5.3 the ZDD based activity-local scheme is directly compared to the ADD based version, by providing ratios of memory consumption concerning Z_R, Z_T and the *peak* number of nodes (r_{pk}), as well as by also giving the ratio of the CPU times consumed for generating the symbolic SG representations (r_{time}). As illustrated by the ratios for the various case studies, the use of ZDDs reduces memory consumption by a factor between 2 and 3. As a a consequence of smaller DD sizes, the caching behavior of the ZDD based scheme is much better. Thus the improvement in run-time is not really surprising.

The TQN model constitutes a very interesting case study, due to the dense enumeration of the states. As one may recall from the previous section, we specified this model as a SPN consisting of 3 places, where two may contain the number of tokens specified by the scaling parameter N, and the remaining place (place 3) contains either one or zero tokens. Consequently, for $N = 2^{n_i} - 1$ the model uses a very dense Boolean enumeration scheme, where n_i is the number of bits used for encoding place $i \in \{1, 2\}$. As we expected, and as supported by the experimental data, in these cases the space requirement of the ADD based

N	$n_{\mathcal{V}}$	(A) ZDD based scheme: nodes Z_R	nodes Z_t	nodes *peak*	t_g	r_{chr}	(B) ADD based scheme: nodes A_R	nodes A_t	nodes *peak*	t_g	r_{chr}
Kanban											
4	96	116	1,902	45,056	0.17	0.38	261	4,898	124,155	0.427	0.10
5	96	163	2,751	83,250	0.39	0.38	321	6,308	208,257	0.882	0.24
6	96	215	3,704	140,709	0.84	0.38	389	7,876	327,853	1.649	0.24
7	96	273	4,758	222,282	1.66	0.37	458	9,521	488,433	2.951	0.26
8	128	341	6,020	333,193	3.26	0.38	731	14,698	920,520	7.392	0.23
9	128	416	7,414	544,593	6.76	0.38	837	17,196	1,277,106	11.442	0.25
10	128	497	8,933	660,963	9.99	0.38	952	19,877	1,719,826	16.999	0.26
FMS											
5	94	391	10784	104218	0.293	0.38	799	26662	265535	0.699	0.14
6	96	559	17557	178566	0.521	0.37	1039	40274	417084	1.159	0.14
7	98	756	26567	284512	0.852	0.37	1298	56853	625228	1.769	0.17
8	118	992	38610	432276	1.409	0.39	2022	96647	1098620	3.155	0.15
9	120	1262	53740	625983	2.062	0.39	2455	129644	1538978	5.395	0.13
10	124	1566	72341	872386	3.067	0.38	2907	167798	2077362	6.942	0.16
Polling											
15	120	103	862	19,257	0.059	0.31	205	2,397	41,164	0.079	0.15
20	160	136	1,252	34,961	0.137	0.32	271	3,567	75,563	0.187	0.14
25	200	171	1,557	53,384	0.260	0.32	341	4,427	113,968	0.309	0.14
TQN											
63	26	19	173	22,967	0.443	0.21	8	243	38,697	0.546	0.18
64	30	16	174	21,285	0.469	0.23	17	293	47,371	0.681	0.20
127	30	22	204	59,221	2.338	0.21	9	286	115,873	2.671	0.16
128	34	18	201	53,979	2.376	0.22	19	338	140,580	3.169	0.21
255	34	25	235	235,400	12.353	0.20	10	329	374,041	13.699	0.12
256	38	20	228	264,709	12.729	0.22	21	383	454,845	15.030	0.16
511	38	28	266	857,570	61.146	0.20	11	372	1,294,609	64.346	0.138
512	42	22	255	814,303	62.907	0.21	23	428	1,584,854	73.724	0.136
FTMP											
2	132	256	5,792	74,834	0.277	0.29	515	13,764	176,259	0.501	0.11
3	196	610	16,225	254,586	1.179	0.29	1213	38,898	588,917	1.874	0.13
4	262	1,044	30,892	620,091	3.268	0.30	2084	74,301	1,433,055	5.897	0.17
5	326	1,556	49,845	974,165	7.257	0.30	3096	119,859	2,824,701	11.087	0.13
6	390	2,146	73,002	2,278,421	14.082	0.30	4269	175,608	5,019,909	20.221	0.12
Courier											
4	144	271	3,490	416,832	3.781	0.34	571	9,255	1,166,600	7.665	0.14
5	144	353	4,715	730,288	5.871	0.35	683	11,475	1,952,685	11.589	0.14
6	144	433	5,941	1,211,540	8.778	0.35	788	13,625	3,140,108	18.076	0.17
7	144	515	7,184	1,942,535	13.493	0.35	894	15,770	4,833,001	24.076	0.18
8	166	603	8,487	2,963,864	20.128	0.36	1196	21,019	8,463,862	40.248	0.16

Table 5.2: Run-time data of the activity-local scheme employing ADDs and ZDDs

representation of the reachable states Z_R is better (see col. r_{Z_R} of Tab. 5.3). If N is a power of two, the enumeration scheme is much sparser and a different picture has to be drawn. Concerning the symbolic representation of the SG Z_T, it is interesting to note that ZDDs are always more compact (see col. r_{Z_T} of Tab. 5.3). Furthermore, the ZDD based scheme maintains its run-time advantage in both cases, which seems to be a consequence of the fact that for ZDDs one does not need to allocate nodes for 0-assigned bit positions within the boolean encoded SVs. This is significant, since the TQN-model constitutes a worst case scenario, since the number of transitions to be explicitly explored and encoded is extremely high (cf. col. $trans_e$ in Tab. 5.1). Therefore node allocation as executed during explicit encoding

Kanban

N	r_{Z_R}	r_{Z_T}	r_{pk}	r_{t_g}
4	2.25	2.58	2.76	2.54
5	1.97	2.29	2.5	2.27
6	1.81	2.13	2.33	1.96
7	1.68	2.00	2.20	1.78
8	2.14	2.44	2.76	2.26
9	2.01	2.32	2.35	1.69
10	1.92	2.23	2.60	1.70

TQN

N	r_{Z_R}	r_{Z_T}	r_{pk}	r_{t_g}
63	0.42	1.40	1.68	1.23
64	1.06	1.68	2.23	1.45
127	0.41	1.40	1.96	1.14
128	1.06	1.68	2.6	1.33
255	0.40	1.40	1.59	1.11
256	1.05	1.68	1.72	1.18
511	0.39	1.40	1.51	1.05
512	1.05	1.68	1.95	1.17

Courier

N	r_{Z_R}	r_{Z_T}	r_{pk}	r_{t_g}
3	1.90	2.41	2.49	1.81
4	2,10	2.65	2.8	2.03
5	1.93	2.43	2.67	1.97
6	1.82	2.29	2.59	2.06
7	1.74	2.2	2.49	1.78
8	1.98	2.48	2.86	2.00

FMS

5	2.04	2.47	2.55	2.38
6	1.86	2.29	2.34	2.23
7	1.72	2.14	2.20	2.08
8	2.04	2.50	2.54	2.24
9	1.95	2.41	2.46	2.62
10	1.86	2.32	2.38	2.26

Polling

15	1.99	2.78	2.14	1.34
18	1.98	2.85	2.26	1.35
20	1.99	2.85	2.16	1.36
21	1.99	2.85	2.25	1.54
25	1.99	2.84	2.13	1.19

FTMP

2	2.01	2.38	2.36	1.81
3	1.99	2.4	2.31	1.59
4	2.00	2.41	2.31	1.80
5	1.99	2.41	2.90	1.53
6	1.99	2.41	2.20	1.44

Table 5.3: Ratios for comparing ADDs and ZDDs

of the different bit-positions of the generated transition, is much more run-time significant than in case of other models.

5.3.2 Assessment of the new symbolic reachability analysis algorithm

For most case studies, the number of explicitly explored and encoded transitions ($trans_e$) under the activity-local scheme is very low (see Tab. 5.1). Therefore, under this scheme, similar to the fully symbolic approaches, most of the execution time is consumed by symbolic reachability analysis. The precise portion of time differs, of course, for different models. For instance, for the Kanban and FTMP models one spends about 70% on symbolic reachability analysis, whereas for the FMS and Courier models symbolic reachability analysis accounts for 99% of the run-time. As a consequence, most of the CPU time is spent in routines for manipulating the DD structures. Profiling reveals that a dominant fraction of the run-time, between 35% and 68%, is spent in the CUDD-functions *UniqueInter* and *CacheLookup*, where other functions consume less than 10%. *UniqueInter* delivers either an existing node found in the "unique table", or a newly allocated node. The *CacheLookup* function accesses the "computed table" in order to fetch results from previous recursions of the *p*ZDD manipulating algorithms. Tab. 5.4.A shows the run-time data of the standard bfs reachability algorithm, Tab. 5.4.B, presents the run-time data of the new quasi-dfs variant and Tab. 5.4.C gives ratios for the run-time (r_t) and the peak number of nodes (r_{pk}), where everything was normed to the figures of the new quasi-dfs reachability analysis algorithm. For evaluating the different reachability algorithms we measured the run-times (t_g), the peak number of nodes allocated (pk), and the number of calls to *UniqueInter* ($c2ut$) and *CacheLookup* ($c2ct$) in millions. As one can read from Tab. 5.4 the new scheme produces fewer calls to *UniqueInter* and *CacheLookup*, making it substantially faster. But on the other hand it consumes more memory as indicated by r_{pk}. In contrast, using ADDs for SG representation, the new scheme is not only faster, but also consumes less memory. We can report this result not only for the ADD based experiments carried out with our Möbius implementation (not shown in the tables), but also for experiments carried out with the tool Caspa (cf. Tab. 5.5).

Tab. 5.5 shows results as obtained from employing the new quasi-dfs reachability analysis and the standard bfs reachability analysis within the tool Caspa, which is based on ADDs. It indicates the peak number of nodes allocated during the SG gen. procedure (*peak*), the number of iterations of the outer `do-until` loops (*iter.*) (cf. Algo. 4.3.A and B, p. 90) and the CPU time consumed by the whole process of generating an ADD based SG representation (t_g). For simplifying the comparison, Tab. 5.5 also gives ratios, where everything was

	(A) Bfs. based RA				(B) Quasi-dfs based RA				Ratios	
N	t_{gen}	$peak$	$c2ut$	$c2cl$	t_{gen}	$peak$	$c2ut$	$c2cl$	r_t	r_{pk}
Kanban										
6	2.91	84,324	4.60	8.98	0.84	140,709	1.01	1.71	3.46	0.60
7	8.62	146,711	13.68	26.45	1.66	222,820	1.93	3.15	5.20	0.66
8	17.60	247,535	24.05	48.59	3.26	333,193	3.44	5.78	5.39	0.74
9	44.17	420,064	58.82	118.78	6.76	544,593	6.06	9.98	6.53	0.77
10	99.68	602,996	136.96	275.64	10.28	660,963	10.11	16.47	9.69	0.91
FMS										
6	0.65	50,419	1.09	2.33	0.52	178,566	0.65	1.49	1.25	0.28
7	1.38	76,357	2.32	4.76	0.85	284,512	1.06	2.36	1.63	0.27
8	3.44	112,877	5.06	11.16	1.41	432,276	1.67	4.06	2.44	0.26
9	7.43	158,074	11.18	24.72	2.06	625,983	2.43	5.87	3.60	0.25
10	15.77	213,728	22.66	49.48	3.07	872,386	3.65	8.84	5.14	0.24
Polling										
15	0.09	9,068	0.12	0.20	0.06	19,257	0.07	0.13	1.46	0.47
20	0.27	16,118	0.29	0.47	0.14	34,961	0.13	0.24	1.99	0.46
25	0.65	23,337	0.66	1.04	0.26	53,384	0.21	0.39	2.50	0.44
Tandem										
64	0.45	10,805	0.82	0.99	0.47	21,285	0.90	1.50	0.96	0.51
128	2.31	31,141	3.57	4.15	2.38	53,979	4.29	4.11	0.97	0.58
256	12.53	161,610	17.68	20.22	12.73	264,709	17.90	19.51	0.98	0.61
512	63.35	710,688	84.21	95.51	62.91	814,303	80.66	89.04	1.01	0.87
FTMP										
2	0.20	27,405	0.28	0.45	0.28	74,834	0.37	0.65	0.71	0.37
3	3.15	71,373	4.77	8.36	1.18	254,586	1.34	2.44	2.67	0.28
4	15.91	153,669	21.62	37.93	3.27	620,091	3.20	6.64	4.87	0.25
5	86.70	286,682	126.18	213.23	7.26	974,165	5.72	12.37	11.95	0.29
6	1035.49	495,192	133.02	2330.20	14.08	2,278,421	12.31	229.40	73.53	0.22
Courier										
4	8.59	121,458	8.60	24.68	3.78	416,832	3.53	6.57	2.27	0.29
5	14.21	228,243	14.29	39.63	5.87	730,288	5.57	12.85	2.42	0.31
6	23.43	497,443	24.89	65.34	8.78	1,211,540	8.33	18.70	2.67	0.41
7	47.02	967,500	53.38	130.63	13.49	1,942,535	12.17	26.59	3.48	0.50
8	121.12	1,922,300	126.85	326.20	20.13	2,963,864	17.62	40.75	6.02	0.65

Table 5.4: Comparison of the two variants of symbolic reachability analysis as implemented within the tool Möbius and by employing ZDDs

normed to the figures of the new quasi-dfs algorithm.
Even though the number of iterations of the outer `do-until` loop of the algorithm (Fig. 4.3.A) is reduced by a factor of about 4 (cf. r_{iter}, Tab. 5.5) under the new scheme, the run-times only halve, and the peak memory requirement is reduced by a factor of around 2 to 3. The moderate speed-up might be a consequence of the very compact encodings of each state by Caspa, since it employs a dense enumeration scheme of submodel states, so that each state requires only a few bits for binary encoding, ultimately leading to "flat" ADD-structures. Thus, it seems that the quasi-dfs scheme becomes more advantageous concerning run-time and space complexity, the larger the generated DDs are. This is not only supported by the figures produced by Caspa, but also by the results obtained under the activity-local scheme, since for larger scaled models the run-time advantages are increasing.

N	Quasi-dfs scheme			Bfs. scheme			Ratios		
	peak	*iter.*	t_{gen}	*peak*	*iter.*	t_{gen}	r_{pk}	r_{itr}	r_t
Kanban									
5	9,820	16	0.170	25,621	71	0.313	2.61	4.44	1.85
6	14,544	19	0.333	47,538	85	0.690	3.27	4.47	2.07
7	18,439	22	0.531	76,384	99	1.310	4.14	4.5	2.47
8	25,840	25	0.892	116,978	113	2.424	4.53	4.52	2.72
9	34,641	28	1.560	168,921	127	3.870	4.88	4.54	2.48
10	48,034	31	2.570	248,749	141	6.710	5.18	4.55	2.61
FMS									
5	33,012	10	0.392	44,854	41	0.617	1.36	4.10	1.58
6	60,337	12	0.948	95,412	49	1.612	1.58	4.08	1.70
7	93,834	14	1.830	167,392	57	3.366	1.78	4.07	1.84
8	136,405	16	3.169	287,628	65	7.299	2.11	4.06	2.3
9	197,777	18	5.700	473,845	73	12.910	2.40	4.06	2.26
10	268,851	20	8.670	728,265	81	23.420	2.71	4.05	2.70

Table 5.5: Comparison of the two variants of symbolic reachability analysis as implemented within the tool Caspa

5.3.3 Significance of variable ordering

It is known that the size of *Mt*-DDs depends on the orderings of the Boolean variables over which the *Mt*-DD is defined. Therefore also the CPU time consumed for traversing the *Mt*-DDs is sensitive to the variable ordering. Thus the variable ordering may severely effect the time for SG generation. In order to illustrate this relation we investigated the Kanban and FMS model, where two different variable orderings were assessed:

(1) In the first setting we employed the variable ordering as chosen by the tool Prism, which basically means that (module)-local variables are grouped together and global variables appear at first, in the order of their appearance within the model specification (grouped ordering).

(2) In the second setting we employed a random variable ordering in case of the FMS model, i.e. the sequence of SVs as produced by the Möbius modeling framework was chosen as variable ordering (random ordering). For the Kanban model, the variable ordering was chosen in such a way that SVs, affected by the same activities were not grouped together, in fact it was tried to put them as far from each other as possible (un-grouped ordering).

Tab. 5.6.A shows the run-time data obtained when generating the SGs for the different orderings and the different models. Tab. 5.6.A records the number of nodes for symbolically representing the set of reachable states and reachable transitions (Z_R and Z_T), the peak number of nodes allocated (*peak*) and the CPU times consumed for generating the symbolic structures (t_g). Additionally Tab. 5.6.B gives the ratios for comparing the figures of the different orderings to each other, where everything was normed to the figures of the grouped ordering.
As one can deduce from the figures of Tab. 5.6, the ordering severely affects the performance of the scheme, where the grouping strategy seems to deliver best results. Given that for the FMS model the activity-local generation spends $\approx 99\%$ of its CPU time on symbolic reachability analysis and given that for the Kanban model this is only $\approx 75\%$, it is not surprising that in case of the FMS model the impact of the variable ordering on the SG generation time is much larger than in case of the Kanban model. –Concerning the experiments carried out with Prism and Caspa, the following results can be reported: (a) By making all SVs of a model specification global, one can also choose an arbitrary order in Prism. Not surprisingly the ungrouped-ordering of the SVs then severely slowed down the performance of Prism's SG generation scheme. (b) The Caspa tool also employs a strategy, where the

(A) Run-time data for comparing the different orderings

N	$n_{\mathcal{V}}$	nodes Z_R	nodes Z_T	nodes *peak*	t_g in sec	nodes Z_R	nodes Z_T	nodes *peak*	t_g in sec
Kanban									
5	96	163	2,751	243,030	0.49	13,560	260,430	2,038,589	14.91
6	96	215	3,704	506,457	1.09	29,036	581,820	4,888,054	30.46
7	96	273	4,758	976,806	2.18	56,224	1,160,790	10,561,445	87.96
		(i) Grouped ordering				(ii) Ungrouped ordering			
FMS									
6	96	559	17,312	195,140	0.29	5,449	153,031	1,161,149	6.23
8	118	992	37,572	436,904	0.78	13,602	469,594	3,774,975	114.70
10	124	1566	69,631	818,852	1.65	27,574	1,125,250	9,405,208	1,105.28
		(i) Grouped ordering				(ii) Random ordering			

(B) Ratios for comparing the different orderings

Kanban

N	r_R	r_T	r_{pk}	r_{time}
5	83.19	94.67	8.39	30.31
6	135.05	157.08	9.65	27.89
7	205.95	243.97	10.81	40.42

FMS

N	r_R	r_T	r_{pk}	r_{time}
6	9.75	8.84	5.95	21.33
8	13.71	12.50	8.64	147.79
10	17.61	16.16	11.49	670.64

Table 5.6: Assessing the significance of the variable orderings

SVs of the submodels are grouped together. Thus Caspa automatically orders the SVs in a grouped fashion, where SVs affected by the same activities appear close to each other, not surprisingly Caspa therefore executes the symbolic SG analysis very efficiently.

In total the above discussion allows one to come to the conclusion that *an ordering of SVs, where SVs manipulated by the same activity are grouped together, is a good heuristic.* This was not only confirmed by our Möbius based experiments, but also by the experiments carried out with Prism and Caspa, and as we recently noticed is also reported in [HKN+03] for being a good strategy.

5.4 Comparison of symbolic SG generation techniques

In this section the activity-local scheme is compared to various other symbolic SG generation techniques.

5.4.1 Comparison to fully symbolic methods

In this subsection the activity-local scheme is compared to the fully symbolic methods as incorporated into the Prism and Caspa tool. This comparison makes sense, since both tools are making use of the CUDD library, which we also use for our implementation of ZDDs. But in contrast to Caspa and as one may recall from the beginning of this chapter, Prism also encodes the different SVs of a model specification individually, thus the focus of the comparison is on PRISM rather than CASPA.

ADD based approach of the tool Caspa

The run-time data as obtained for different benchmark models analyzed with our Möbius implementation or with the tool Caspa are given in Tab. 5.7. The pure data is given in subtable (A), containing the SG generation time t_g, the peak number of nodes *peak*, the number of nodes for representing the SG Z_T and the number of boolean variables $n_{\mathcal{V}}$ employed for encoding the SG of the resp. model. Tab. 5.7.B contains ratios, where the data

(A) Run-time data

N	Caspa				Act.-local			
	t_g	# ADD nodes:		$n_\mathcal{V}$	t_g	# ZDD nodes:		$n_\mathcal{V}$
		peak	Z_T			*peak*	Z_T	
Kanban								
6	0.940	47,395	5,164	56	0.840	140,709	3,704	96
7	1.590	76,230	6,776	56	1.660	222,282	4,758	96
8	3.020	116,785	9,138	64	3.260	333,193	6,020	128
9	4.870	168,694	11,587	64	6.760	544,593	7,414	128
10	8.370	248,461	15,158	72	9.990	660,963	8,933	128
TQN								
63	0.090	3,328	235	26	0.443	2,467	173	26
255	1.940	22,967	321	34	12.353	235,400	235	34
511	6.930	24,241	364	38	61.146	85,760	266	38
1023	37.560	47,909	407	42	278.365	368.768	298	42
FMS								
6	1.930	95,318	19,534	50	0,521	178,566	17,557	96
7	4.020	167,265	30,670	56	0,852	284,512	26,567	98
8	7.670	289,445	46,892	62	1,409	432,276	38,610	118
9	14.810	469,258	69,006	68	2,062	625,983	53,740	120
10	30.860	735,203	96,216	72	3,067	872,386	72,341	124
Polling								
15	0.050	9,844	1,942	40	0.059	19,257	862	120
20	0.170	20,252	3,345	52	0.137	34,961	1,252	160
21	0.210	23,258	3,674	54	0.156	42,150	1,314	168

(B) Ratios

N	r_{t_g}	r_{peak}	r_{Z_T}	$r_{n_\mathcal{V}}$
Kanban				
6	1.119	0.337	1.394	0.583
7	0.958	0.343	1.424	0.583
8	0.926	0.351	1.518	0.500
9	0.720	0.310	1.563	0.500
10	0.838	0.376	1.697	0.563
TQN				
63	0.203	1.349	1.358	1.000
255	0.157	0.098	1.366	1.000
511	0.113	0.283	1.368	1.000
1023	0.135	0.130	1.366	1.000

N	r_{t_g}	r_{peak}	r_{Z_T}	$r_{n_\mathcal{V}}$
FMS				
6	3.707	0.534	1.113	0.521
7	4.720	0.588	1.154	0.571
8	5.442	0.670	1.215	0.525
9	7.181	0.750	1.284	0.567
10	10.061	0.843	1.330	0.581
Polling				
15	0.847	0.511	2.253	0.333
20	1.241	0.579	2.672	0.325
21	1.346	0.552	2.796	0.321

Table 5.7: Comparing the activity-local scheme to Caspa

of the activity-local scheme served as norm.

The run-time data and thus the comparison with Caspa, which employs a fully symbolic SG generation scheme has to be considered with care, since:

(1) In Caspa the status of a process is recorded by a single SV (denoted as state counter), rather than a set of SVs, yielding a much denser enumeration scheme of states (cf. $n_\mathcal{V}$ in Tab.5.7.A).

(2) The ordering of the boolean vectors encoding the individual state counters depends on the occurrence of the submodel description within the overall model specification, clearly influencing the efficiency of the SG generation routines (M. Kuntz pers. com.)

(3) Caspa employs ADDs, which are known to be often more memory and run-time consuming than our ZDDs. It is surprising to note, that even though Caspa requires fewer Boolean variables for encoding each state, ZDDs are still more memory efficient (cf. col. r_{nv} and $r_{\mathbb{Z}_T}$ in Tab. 5.7.B).

(4) Caspa makes use of the standard bfs symbolic reachability analysis, where the activity-local scheme employs the new quasi-dfs scheme, leading to significant run-time advantages in case of the latter.

(5) Caspa's input language is a stochastic process algebra emphasizing compositional model descriptions. On the level of the symbolic representation, the individual SGs of the processes are composed via synchronization. Thus the efficiency of the SG generation scheme highly depends on the compositional character of the model description (M. Kuntz pers. com.). In case one would for instance describe a model as a complex single process the scheme for generating a symbolic SG representation would turn out to be not efficient. For exemplification one may refer to the Kanban and FMS model. Since the Kanban model possesses a nice compositional structure, its SG can be generate very quickly. In contrast to this, the FMS model does not possess such a nice structure, thus its SG generation consumes more CPU time.

According to the above points, it is clear that for models incorporating a compositional structure, Caspa's fully symbolic SG generation scheme may be more efficient than our activity-local scheme. However, the new data type in combination with the new algorithm for symbolic reachability analysis often compensates the disadvantages as arising from the (sometimes extensive) explicit SG generation as executed by the activity-local scheme. For exemplification one may refer to the ratios of the Kanban and TQN model in Tab. 5.7.B. The activity-local scheme requires an extensive explicit exploration of activities, yielding substantially larger run-times for generating a symbolic SG representation. Hereby especially those activities play a crucial role, which are employed within Caspa for synchronization. This is because within a monolithic model description, these synchronizing activities are modeled as a single activity, yielding often large sets of dep. SVs and activities. As a consequence, the obtained (complex) activity needs to be explored explicitly for all different values its dependent SVs can take. Within a partitioned model, the (complex) activity is distributed among independent submodels. The sum of the individual executions of the obtained submodel-local activities within the submodels is then much lower than the total number of executions of the non-decomposed activity within a monolithical model description.

For the FMS and highly scaled Polling models the situation differs. Here the model can either not be nicely partitioned and/or activities are only connected to few SVs and all SVs are one-bounded. Consequently the explicit exploration of activities is limited to a few transitions and the activity-local scheme leads to much smaller SG generation times than the purely symbolic approach of the tool Caspa (cf. col. r_{t_g} of Tab. 5.7.B).

The above discussion makes clear that a good performance of Caspa is closely related to well-partitioned models. This is the case not only for the reason that compositionality reduces the number of iterations when generating the sub-model-local SG representations. It also depends on the ordering of the SVs, where a compositional modeling formalism naturally supports a grouped ordering of SVs. In case of unstructured model specification Caspa does not perform very well and it also does not apply any heuristics for ordering the SVs appropriately. As a matter of fact, in such settings a good performance is then only a matter of luck or the know-how of the modeler, who orders the SVs within a model specification in a grouped fashion.

ADD based approach of the tool Prism

Like Caspa, the tool Prism employs (a) an input language supporting the modular specification of a system, (b) ADDs for the symbolic SG representation, (c) a fully symbolic method for generating the SG of a high-level model, and (d) a standard bfs symbolic reachability analysis for restricting the set of potential transitions to the actually reachable ones.

Tab. 5.8.A gives the basic run-time data of the Kanban, FMS and Polling model, when analyzed with Prism. It records the number of Boolean variables employed by Prism for encoding each state ($n_{\mathcal{V}}$), the number of nodes allocated for an ADD based representation of (a) the set of reachable states (Z_R), (b) the set of reachable transitions (Z_T), and (c) the peak number of nodes allocated during the process of SG generation (*peak*). Furthermore Tab. 5.8.A gives also the number of iterations (*iter*) as required by the standard bfs symbolic reachability analysis (the `do-until` loop of Algo. 4.3.A) and the CPU time consumed for generating an ADD based SG representation.

For the comparison of our approach to Prism, we investigate three different settings as shown in Tab. 5.8.B:

(1) In the first setting we combined the activity-local scheme with ADDs and standard bfs reachability analysis. (col. 4 - 7 of Tab. 5.8.B).

(2) In the second setting the activity-local scheme was enriched with the new algorithm for carrying out the symbolic reachability analysis in a quasi-dfs like style (col. 8 and 9 of Tab. 5.8.B).

(3) In the third setting the standard ADD structure was replaced by the new ZDD data structure (col. 10-14 of Tab. 5.8.B).

The ratios as obtained when norming the Prism run-time data to the figures as produced by the different settings are shown in Tab. 5.8.B. Column one gives hereby the model scaling parameter N, col. 2 contains the number of boolean variables for encoding each transition when the activity-local scheme was employed. Since in case of the FMS - and Polling model the specification differed, Tab. 5.8.B also contains the ratio of the number of boolean variables, where the number of variables as employed by Prism was divided by the number as employed by the activity-local scheme.

As already reported, Prism, similar to our own implementation, encodes the SVs of a model specification individually. Consequently the activity-local scheme is capable of generating isomorphic ADDs to the ones generated by Prism, since the activity-local scheme allows a freely chosen ordering of the Boolean vectors encoding the different SVs. I.e. employing the grouped ordering of SVs combined with an ADD based SG representation and an adequate model specification, one ends up with the very same ADDs for representing the set of reachable states and transitions. This was truly the case for the Kanban - model. However, in case of the FMS and Polling model slightly different high-level model specifications were employed, so that the ADD generated were not isomorphic here. In the following the models will be discussed individually in order to pin-point strength and weakness of the activity-local scheme if compared to Prism.

Kanban model

As already reported above, the input language of Prism supports a modular description of a system, where the submodels are composed via activity synchronization. In order to investigate the sensitivity of the SG generation scheme with respect to a modularity of a model specification, we also studied a Kanban model specification consisting of a single partition (Kanban 1-P, grouped). As one can see in Tab. 5.8.A, the re-arrangement of the Kanban model does not affect the SG generation very much. The consistently better run-times of the monolithic model description might be due to the circumstance that no activity synchronization took place and no administering of different partitions had to be done. This

(A) Run-time data as produced by the tool Prism

N	$n_{\mathcal{V}}$	ADD nodes: Z_R	ADD nodes: Z_T	ADD nodes: *peak*	*iter*	t_g
Kanban (4 partitions, grouped ordering)						
6	96	389	7,876	93,464	85	1.492
7	96	458	9,521	135,514	99	2.386
8	128	731	14,702	246,750	113	4.997
9	128	837	17,196	333,388	127	7.910
10	128	952	19,877	437,910	141	13.780
Kanban (1-P, grouped ordering)						
6	96	389	7,876	93,464	85	1.34
8	128	731	14,702	246,750	113	4.255
10	128	952	19,877	437,910	141	9.999
Kanban (1-P, ungrouped ordering)						
6	96	389	7,876	4,785,597	85	111.14
8	128	???	???	???	113	863.13

N	$n_{\mathcal{V}}$	ADD nodes: Z_R	ADD nodes: Z_T	ADD nodes: *peak*	*iter*	t_g
FMS						
6	110	1,383	45,984	211,372	49	2.732
7	110	1,731	56,141	330,636	57	5.558
8	140	2,770	110,237	643,698	65	22.552
9	140	3,308	143,374	929,050	73	33.468
10	140	3,900	176,329	1,304,115	81	65.431
Polling						
15	40	47	1,942	12,796	31	0.161
20	52	63	3,346	27,768	41	0.338

(B) Ratios

N	$n_{\mathcal{V}}$	$r_{n_{\mathcal{V}}}$	ADDs and std. symb. reach. algo. r_{Z_R}	r_{Z_T}	r_{peak}	r_{tg}	ADDs and new reach. r_{peak}	r_{tg}	ZDDs and new symb. reach. algo. r_{Z_R}	r_{Z_T}	r_{peak}	r_{tg}
Kanban (4 partitions, grouped ordering)												
6	96	1.00	1.00	1.00	0.05	0.06	0.29	0.90	1.81	2.13	0.66	1.77
7	96	1.00	1.00	1.00	0.04	0.03	0.28	0.81	1.68	2.00	0.61	1.44
8	128	1.00	1.00	1.00	0.04	0.03	0.27	0.68	2.14	2.44	0.74	1.53
9	128	1.00	1.00	1.00	0.03	0.05	0.26	0.69	2.01	2.32	0.61	1.17
10	128	1.00	1.00	1.00	0.03	0.05	0.25	0.81	1.92	2.23	0.73	1.34
FMS												
6	96	1.15	1.33	1.14	0.10	0.39	0.51	2.36	2.47	2.62	1.18	5.25
7	98	1.12	1.33	0.99	0.07	0.42	0.53	3.14	2.29	2.11	1.16	6.53
8	118	1.19	1.37	1.14	0.06	0.75	0.59	7.15	2.79	2.86	1.49	16.00
9	120	1.17	1.35	1.11	0.05	0.61	0.60	6.20	2.62	2.67	1.48	16.23
10	124	1.13	1.34	1.05	0.04	0.68	0.63	9.43	2.49	2.44	1.49	21.33
Polling												
15	120	0.33	0.23	0.81	0.63	2.12	0.27	1.92	0.45	2.25	0.40	3.66
20	160	0.33	0.23	0.94	0.69	2.06	0.32	1.97	0.46	2.15	0.47	4.02

Table 5.8: Comparing the activity-local scheme to Prism

speaks first of all to the advantage of the Prism tool. However, on the other hand, this effect has also to do with the fact, that the chosen orderings of the SVs for both model specification were identical. Since symbolic reachability seems to be the main source of CPU time consumption for Prism, it is clear that Prism's SG generation time is highly sensitive to the chosen ordering of the SVs, especially in the presence of global variables. Thus it is not very surprising that under an un-grouped ordering the non-partitioned Kanban model (Kanban 1-P, un-grouped in Tab. 5.8.A) the run-time as well as the peak number of ADD nodes allocated increases dramatically, making the analysis of the Kanban model for large

scaling parameters cumbersome or even not possible. Already for a scaling parameter $N := 8$, we can report that Prism failed to count any numbers of nodes allocated, since it simply crashed after generating a symbolic representation of the SG (cf. Kanban 1-P model in Tab. 5.8.A). This clearly illustrates why a comparison under a fixed SV ordering is essential. It furthermore makes clear that the good performance of Prism in case of the Kanban model stems from the fact that the Kanban model is highly suited for compositional approaches and that it naturally supports the grouped ordering of SVs. In case of unstructured model specification and in the presence of global variables Prism does not apply any heuristics for ordering the SVs. Thus in such cases a good performance when generating a symbolic representation of a model's SG is only a matter of luck or the know-how of the modeler, who orders them in a grouped fashion.

FMS model
The manual elimination of immediate activities within the high-level model description allowed us to encode each transition with a smaller number of SVs, if compared to the model analyzed by Prism (col. $r_{n\mathcal{V}}$ in Tab. 5.8.B for the FMS model). As a consequence the obtained ADDs were already slightly smaller than the ones produced by Prism (cf. col. 4 and 5 of Tab. 5.2.B, which contain the ratios for the number of nodes of Z_R and Z_T). As one can see, these differences are very small. One may conclude now that the moderately worse performance of the activity-local scheme is due to the more compact encoding of transitions. However, this is to be doubted, since if one takes a look at the FMS model, it appears that the number of dep. SVs of each activity is relatively small. Thus the number of transitions to be explored explicitly is very low as well (col. $trans_e$ Tab. 5.1). As a consequence of this, most of the CPU-time consumed by the activity-local scheme is also spent for symbolic reachability analysis. This shift towards symbolic reachability analysis decreases the time advantages of the fully symbolic methods, since for them symbolic reachability analysis is also the main resource of CPU time consumption. This point of view is also supported by the run-time data of the Polling model to be discussed next.

Polling model
In contrast to the Kanban and FMS model, the run-time data of the Polling model speaks solely to the advantage of the activity-local scheme. This is even more remarkable, since (a) the number of boolean variables for encoding each transition is significantly higher (col. $r_{n\mathcal{V}}$ in Tab. 5.8.B for the Polling model) and (b) the number of ADD nodes is significantly higher, and (c) the new symbolic reachability algorithm does not improve the run-times. Thus the better performance of the activity-local scheme might stem from the circumstance that the Polling model consists of many activities, but each is connected to very few 1-bounded SVs. The concurrency of the activities is therefore the main source for the SG explosion and not a large number of tokens appearing in the various places, as in case of the Kanban model. However, with the activity local scheme concurrency is only extracted on the level of symbolic SG composition, rather than on the level of explicit exploration. As a consequence the number of transitions to be explicitly explored is very low ($trans_e$ in Tab. 5.1) so that the overall SG generation time is highly competitive.

The above discussion allows us to draw the following conclusions: Explicit handling of transitions induces a non-negligible run-time overhead for the activity-local scheme in comparison to Prism's fully symbolic method, which does not execute any explicit SG generation (cf. col. 7 (r_{tg}) of Tab. 5.8.B). However, this overhead is justified by two aspects:

(1) The activity-local approach, in contrast to Prism, is not restricted to any specific model description method, which is of great importance for tools relying on multiple formalisms such as Möbius.

(2) Monolithic models or models not suitable for being partitioned and composed via activity-synchronization (such as the FTMP and the FMS model) can be analyzed very efficiently, where submodel-oriented approaches are problematic.

As shown by the last 6 columns of Tab. 5.8.B, the new scheme for reachability analysis as well as the use of ZDDs improves the run-time significantly.

As with all symbolic representation techniques, memory space is not an issue. But nevertheless a comparison with Prism shows a significantly large peak number of nodes to the disadvantage of the activity-local scheme (cf. col. 6 of Tab. 5.8.B). But our implementation stores redundant *Mt*-DDs in order to simplify and speed up the whole scheme (cf. Tab. 5.2.A), if memory was at a premium, the redundancy could easily be eliminated without a dramatic increase in run-time. The storage of the sets of activity-local state markings which were already tested for new model behavior, is not necessary. Instead this information could be extracted from the activity-local transition systems. The redundant testing of the enabledness of activities would hereby induce only a small run-time overhead. But nevertheless, even if storing redundant structures, the use of ZDDs already improves the situation significantly (col. 12 of Tab. 5.8.B).

5.4.2 Comparison to semi-symbolic methods

In this section the activity-local scheme, which belongs to the compositional, semi-symbolic and sub-model-interdependent schemes, is not only compared to a semi-symbolic, monolithic method, but also to other semi-symbolic, compositional methods, where the more sophisticated ones are based on the well known saturation technique [CLS01].

Comparison to a non-compositional semi-symbolic method

With a non-compositional semi-symbolic method all transitions are generated and encoded explicitly. Thus methods of this kind suffer from extremely long run-times and an increase in the peak memory sizes, where the latter is even worse when ZDDs are employed. Tab. 5.9 illustrates this behavior for different models. As basic data its first seven columns show the model scaling parameter (N), number of states ($states$), the number of transitions ($trans$), the number of transitions as explored under the activity-local scheme ($trans_e$), the number of boolean variables required for encoding each transition ($n_{\mathcal{V}}$), as well as the number of ZDD-nodes allocated for representing the set of reachable states (sz_{Z_R}) and the set of reachable transitions (sz_{Z_T}). For comparing the monolithic generation scheme with the activity-local approach the last 4 columns of Tab. 5.9.A show the peak number of nodes and CPU times as consumed by the different approaches during SG generation. For simplification Tab. 5.9.B gives their ratios, where everything was normed to the data of the activity-local scheme. The performance of the monolithic procedure is hereby so poor, that the benchmark models could only be analyzed for very small scaling parameters.

Comparison to other compositional semi-symbolic methods

In the following sections the activity-local scheme is compared to the MDD based approach of [DKS03], which is also implemented within Möbius. Thus our implementation and the implementation of [DKS03] uses the same Möbius high-level model specification and the standard next-state function of Möbius' interface for explicit SG exploration. The activity-local scheme will also be compared to the MDD and *KO* based approach of the tool Smart, due to the numerous symbolic approaches implemented there [CM99b, CLS01, CMS03] and due to its broad acceptance. However, one may already note that in both cases and contrary to the activity-local approach, the effectiveness of the employed compositional semi-symbolic SG techniques, i.e. the approaches of [DKS03] and [CM99b, CLS01, CMS03] heavily depend on well partitioned models, where the partitioning needs to be specified by the user.

MDD based approach of [DKS03]

In order to compare the activity-local scheme to the symbolic approach of [DKS03], we analyzed the Courier, Polling and FTMP model, where for both approaches the same Möbius model specifications were used. We did not analyze the Kanban and FMS model, since in

(A) Model statistics and run-time data

							Monolithic		Act.-local	
N	*states*	*trans.*	$trans_e$	$n\mathcal{V}$	sz_{Z_R}	sz_{Z_T}	*peak*	t_{gen}	*peak*	t_{gen}
Kanban										
3	58,400	446,400	252	64	75	1,176	6,436,298	13.493	39,516	0.036
4	454,475	3,979,850	740	96	116	1,902	71,979,062	177.467	105,621	0.112
FMS										
4	35,910	237,120	180	90	252	5,968	4,097,002	8.061	64,971	0.084
5	152,712	1,111,480		94	391	10,707	20,305,487	50.475	118,914	0.160
Polling										
10	15,360	89,600	40	80	69	558	1,474,140	2.600	14,479	0.012
14	344,064	2,695,170	60	112	97	802	47,310,670	156.798	32,249	0.028
Courier										
1	47,232	206,112	34		71	686	2,640,451	8.913	43,177	0.112
2	434,304	434,304	59	122	126	1,404	40,525,676	147.153	141,585	0.536
FTMP										
1	414	1,656	172	66	82	722	90,895	0.112	11,209	0.012
2	256,932	256,932	688	132	257	5,793	93,689,939	442.172	84,012	0.140

(B) Ratios

N	r_{peak}	$r_{t_{gen}}$
Kanban		
3	162.88	374.80
4	681.48	1,584.39
Courier		
1	61.15	79.57
2	286.23	274.52

N	r_{peak}	$r_{t_{gen}}$
FMS		
4	63.06	95.96
5	170.76	315.45
FTMP		
1	8.11	9.33
2	1,115.20	3,158.15

N	r_{peak}	$r_{t_{gen}}$
Polling		
10	101.81	216.68
14	1,467.04	5,599.93

Table 5.9: Comparison to a non-compositional semi-symbolic SG gen. scheme

contrast to the three afore mentioned models for these models a(n) (efficient) partitioning under the *Join*-composition turned out to be cumbersome and out of scope of this work.

Tab. 5.10 presents the run-time data for the FTMP, Courier and Polling model when employing the MDD based approach of [DKS03] or the ZDD based activity local scheme. Tab. 5.10 gives the memory required for storing the set of reachable states (mem_{Z_R}), and the transition rate matrices (mem_{Z_T}) in KBytes, where the peak memory consumption (*peak*) is given in MBytes. Since the latter is not reported by the software directly, we employed the following approximation (S. Derisavi pers. com):

$$\begin{aligned} peak := {} & peak\_num\_of\_nodes(\mathsf{Z}_R) \cdot (mem_{\mathsf{Z}_R}/num\_of\_nodes(\mathsf{Z}_R)) + \\ & peak\_num\_of\_nodes(\mathsf{Z}_T) \cdot (mem_{\mathsf{Z}_T}/num\_of\_nodes(\mathsf{Z}_T)) \end{aligned}$$

Besides the memory consumption, Tab. 5.10 gives also the number of transitions explicitly explored $trans_e$ and the overall CPU time consumed for generating a symbolic SG representation. For simplifying the comparison we also computed ratios, where everything is normed to the figures of the activity-local scheme. Tab. 5.11 presents the ratios for the memory employed in case of the set of reachable states (r_{Z_R}), the transition rate matrices (r_{Z_T}), and the peak memory consumption (r_{peak}). Most important Tab. 5.11 presents also the run-times of the SG generation schemes (r_{tg}). As an example, the last entry in the last row of Tab. 5.11 states that our activity-local approach is 590.42 times faster. The fact that the activity-local

N	Approach of [DKS03]					Act.-local scheme				
	$memz_R$	$memz_T$	$peak$	$trans_e$	t_g	$memz_R$	$memz_T$	$peak$	$trans_e$	t_g
Courier										
4	31.02	97.81	0.821	103,732	3.204	4.13	52.78	9.16	142	2.548
5	67.63	237.93	2.188	312,478	11.261	5.39	71.77	16.17	206	3.952
6	134.57	523.8	5.221	829,495	38.878	6.67	91.28	26.88	289	6.016
7	254.41	1059.39	11.537	1,983,112	123.716	8.00	111.38	43.30	394	8.985
8	452.95	1996.62	23.498	4,806,745	371.795	9.44	132.63	67.61	525	13.789
Polling										
15	2.51	14.32	0.029	60	0.012	1.63	13.48	0.49	60	0.032
18	3.02	20.19	0.039	72	0.016	1.92	17.67	0.73	72	0.044
20	3.35	24.67	0.046	80	0.028	2.14	19.58	0.90	80	0.056
21	3.52	27.07	0.050	84	0.036	2.25	20.53	0.99	84	0.072
25	4.17	37.8	0.067	100	0.048	2.69	24.34	1.38	100	0.100
FTMP										
2	4.54	5.08	0.109	70,512	1.100	4.02	90.52	1.28	688	0.136
3	10.3	8.12	0.383	601,936	21.505	9.55	253.53	4.65	1,548	0.572
4	17.27	11.16	0.916	2,610,256	166.278	16.33	482.70	11.43	2,752	1.560
5	25.45	14.21	1.770	8,000,496	870.878	24.33	778.84	23.24	4,300	3.276
6	34.84	17.25	3.058	19,847,072	3528.589	33.55	1,140.67	41.50	6,192	5.976

Table 5.10: Comparison to the approach of [DKS03] (run-time data)

approach is significantly faster than the MDD based approach, when analyzing the FTMP model, shows that the partial-exploration strategy by considering model-inherent structures only clearly pays off (cf. cols. $trans_e$ of Tab. 5.10). In contrast, as already mentioned before, the efficiency of the approach of [DKS03] depends on the compositional structure of the high-level model specification. Consequently non-structured models or models requiring a lot of interaction among the submodels, lead to a significant increases in run-time for generating the underlying SG. Both, the Courier and even more the FTMP model seem to be models of such a kind, so that the method of [DKS03] requires to explore a large number of transitions explicitly for generating a symbolic representation of their SGs. Therefore especially these models nicely illustrate the advantages of the activity-local approach, which does not require any particular model structure and is still capable of generating their SGs by consuming considerably less CPU time.
In case of the Polling model a different picture has to be drawn. Here the join based composition works very well. A closer look reveals that each component has a one-bounded input place and output place, where the exchange of tokens only affects the neighboring components. As a consequence, the method of [DKS03] efficiently saturates the SGs of the submodels, and thus generates a MDD based representation of the overall SG by consuming remarkably little CPU time. I.e., as shown by the data of Tab. 5.10, the number of transitions fired is identical for the method of [DKS03] and our activity-local approach. The larger amount of CPU time for each firing and encoding in case of the activity-local approach is not surprising, since the dynamic sets of dependent enabled activities and the individual sets of tested states for each activity need to be updated. In contrast, the method of [DKS03] employs here the static sets of activities of the submodels and stores reached states in a submodel-wise and not in an activity-wise manner. Furthermore the activity-local scheme needs to carry out a symbolic reachability analysis, where [DKS03] generates the set of reachable states by applying the saturation-technique, which works here very well.

As one can deduce from Tab. 5.11, the memory requirement for storing the set of reachable states and the transition rate matrix is most of the times better when employing ZDDs. However, in case of the peak memory requirement it turns out that the method of [DKS03] is more efficient. But as already discussed above, memory space for the ZDD based method

N	r_{Z_R}	r_{Z_T}	r_{peak}	r_{trans}	r_{tg}
Courier					
4	7.52	1.85	0.09	730.51	1.26
5	12.55	3.32	0.14	1,516.88	2.85
6	20.17	5.74	0.19	2,870.22	6.46
7	31.80	9.51	0.27	5,033.28	13.77
8	48.00	15.05	0.35	9,155.70	26.96
Polling					
15	1.54	1.06	0.06	1.00	0.38
18	1.57	1.14	0.05	1.00	0.36
20	1.57	1.26	0.05	1.00	0.50
21	1.56	1.32	0.05	1.00	0.50
25	1.55	1.55	0.05	1.00	0.48
FTMP					
2	1.13	0.06	0.09	102.49	8.09
3	1.08	0.03	0.08	388.85	37.59
4	1.06	0.02	0.08	948.49	106.58
5	1.05	0.02	0.08	1,860.58	265.82
6	1.04	0.02	0.07	3,205.28	590.42

Table 5.11: Comparison to the approach of [DKS03] (ratios)

is not an issue, even though our implementation stores redundant structures. If memory was at a premium, the redundancy could easily be eliminated.

MDD based approaches of Smart
Tab. 5.12 and 5.13 show the run-time data as obtained when generating the SGs for the Kanban, FMS and Polling model with the Smart tool as well as with the the activity-local scheme. Since the Smart tool supports the use of various symbolic (saturation based) SG generation techniques, experiments had to be repeated for the different methods:

(1) saturation un-bounded [CMS03],

(2) saturation based on pre-generation of the local SGs [CLS00] and

(3) pre-generative symbolic approach of [CM99b].

Tab. 5.12 shows the CPU time consumed for generating the SGs of the different models under the different symbolic methods (t_g), it also gives ratios, were the figures of the activity-local scheme served as norm (r_{t_g}). As already pointed out in Sec. 4.7, the methods incorporated into the Smart tool depend (at least) on a "good" partitioning of the overall model, where the partitioning must have *KO compliant* structure. In case one is enabled to find such a partitioning, the methods show very efficient behavior (cf. runtimes of the Kanban model (4 partitions) as presented in Tab. 5.12). In case one fails to give an adequate partitioning, the performance of the methods drops dramatically. In order to study this effect in greater detail, as we already did in case of the Prism tool, we partitioned the Kanban system additionally into 6 partitions. In this latter case, addressed as Kanban model (6-partitions), the partitions were chosen in such a way that the SVs connected to one of the two activities, having more than two dependent SVS, are encapsulated in their own partition. This yields two extra partitions, each containing one of those two activities, which are normally split when applying a *Sync* driven decomposition. As a consequence the remaining four partitions now contain only one or two local SVs, which decreases the number of partition-local transitions. Not surprisingly, this procedure dramatically increases the run-times of the SG generation methods as employed within the Smart tool. As a consequence the activity-local scheme clearly performs better, where it is also independent of a user-defined partitioning. This

	Methods incorprated into Smart						ZDD based
	Saturation.		Sat.-PreGen		Pregeneration		Act.-Local
N	t_g	r_{t_g}	t_g	r_{t_g}	t_g	r_{t_g}	t_g
Kanban (4 partitions)							
6	0.02	0.04	0.02	0.04	0.25	0.45	0.564
7	0.04	0.03	0.04	0.03	0.31	0.27	1.160
8	0.06	0.02	0.05	0.02	0.39	0.14	2.712
9	0.09	0.02	0.07	0.01	0.49	0.09	5.316
10	0.14	0.02	0.1	0.01	0.64	0.07	8.921
Kanban (6 partitions)							
6	4.25	7.54	2.17	3.85	161.9	287.04	0.564
7	7.45	6.42	4.84	4.17	334.14	288.04	1.160
8	14.54	5.36	9.89	3.65	675.28	248.98	2.712
9	30.06	5.65	19.72	3.71	1367.37	257.20	5.316
10	54.36	6.09	37.05	4.15	2655.19	297.65	8.921
FMS							
6	2.74	9.40	13.99	48.04	6.21	21.32	0.291
8	24.81	31.54	23.58	29.99	42.14	53.58	0.786
10	221.09	133.83	53.96	32.66	xxx	xxx	1.652
12	1884.85	561.60	117	34.86	xxx	xxx	3.356
Polling							
10	0.04	1.52	0.33	11.34	15.86	548.44	0.029
15	0.07	1.79	0.49	12.87	54.24	1430.47	0.038
20	0.09	1.05	0.65	7.77	129.58	1552.79	0.083
25	0.11	0.81	0.81	5.86	249.12	1805.64	0.138

Table 5.12: Comparing activity-local scheme and Smart (run-times)

advantage for the activity-local scheme is even more clear in cases, where the finding of a *KO compliant* partitioning is not trivial. In case of the FMS model, we were not able to find an adequate *KO compliant* partitioning, yielding better run-time results than the non-partitioned model, –we did not consider a changing of the high-level model's structure. As a consequence the activity-local approach even more clearly outperforms Smart and its symbolic techniques. In case of the Polling model and the saturation unbounded method, the situation is once again balanced. For similar reasons as discussed in Sec. 5.4.1, the activity-local approach can compete even with well-partitioned model descriptions. The bad performance of the symbolic methods depending on pre-generation of the SGs stems from the fact that the bounds of the SVs must be modeled by either disconnecting places and activities if there are enough tokens in the post set or by making use of inhibitor arcs. If this is not done carefully, the pre-generation techniques will generate local SGs which are significantly larger than the actually reachable ones, inducing a significant overhead on the SG generation time. Tab. 5.13 shows the memory consumption when the different symbolic techniques are employed.[3] I.e. it records the peak memory consumption (*peak*) in MByte, as well as the number of KBytes required for storing the set of reachable states and the set of reachable transitions ($mem_{\mathbf{Z}_R}$ and $mem_{\mathbf{Z}_T}$). It appears that in case of well partitioned models, e.g. Kanban and Polling, the symbolic SG generation methods incorporated into Smart have lower peak memory sizes and require less memory for representing the set of reachable transitions. This might be due to the compactness of a *KO* based representation of the transition rate matrix. In case of the memory required for representing the set of reachable states MDDs and ZDDs deliver similar results. This good behavior of the *KO* based methods of the Smart tool can not be employed for models, where a partitioning is not feasible. It is therefore not surprising that in case of the FMS model the ZDD based activity-local scheme delivers much better results.

[3] In case of positions filled with question marks, Smart simply did not report the required figure.

N	$peak$ (MByte)	mem_{Z_R} (KByte)	mem_{Z_T} (KByte)	$peak$ (MByte)	mem_{Z_R} (KByte)	mem_{Z_T} (KByte)
Kanban	Smart: MDD Sat. (unbounded)			Smart: MDD Sat. (PreGen)		
7	0.0219	3.19	26.56	0.0429	5.37	25.21
8	0.0345	4.50	37.46	0.0713	7.97	35.41
9	0.0523	6.16	51.02	0.1133	11.43	48.10
10	0.0768	8.25	67.52	0.1732	15.91	63.57
	ZDD based act.-local scheme			Smart: PreGen of Local SGs		
7	14.9049	4.27	74.34	0.0042	0.48	24.47
8	28.5925	5.33	94.06	0.0060	0.62	34.66
9	50.9460	6.5	115.84	0.0084	0.79	47.36
10	85.4268	7.77	139.58	0.0115	0.99	62.83
FMS	Smart: MDD Sat. (unbounded)			Smart: MDD Sat. (PreGen)		
6	0.0742	75.19	1670.69	0.0742	75.19	1670.69
8	0.3796	387.34	9415.43	0.3796	387.34	9415.43
10	1.4705	1503.49	38847.44	1.4705	1503.49	38847.44
12	4.6744	4782.99	129197.53	4.6744	4782.99	129197.53
	ZDD based act.-local scheme			PreGen of Local SGs		
6	2.9776	8.73	270.5	0.0368	?	1596.09
8	6.6666	15.5	587.06	0.1892	?	9028.67
10	12.4947	24.47	1087.98	xxx	xxx	xxx
12	20.9370	35.52	1808.94	xxx	xxx	xxx
Polling	Smart: MDD Sat. (unbounded)			Smart: MDD Sat. (PreGen)		
10	0.0015	0.83	3.79	0.1091	57.37	482.81
15	0.0023	1.26	5.68	0.1681	87.57	725.39
20	0.0031	1.69	7.58	0.2271	117.76	968.75
25	0.0039	2.12	9.47	0.2860	147.96	1212.89
	ZDD based act.-local scheme			Smart: PreGen of Local SGs		
10	0.2209	1.08	8.72	0.0139	13.46	482.81
15	0.4921	1.63	13.48	0.0422	21.23	725.39
20	0.8974	2.14	19.58	0.0574	29.00	968.75
25	1.3806	2.69	24.34	0.0032	1.62	1198.54

Table 5.13: Comparing activity-local scheme and Smart (memory consumption)

5.5 Assessing the ZDD based solvers

Tab. 5.1 gives the sizes of the CTMCs of the benchmark models. The experiments of Tab. 5.14.A were carried out on a Pentium IV with 3 GHz, 1 GByte of RAM and a Linux OS. All other results presented in this section were collected on a Pentium IV 2.88 GHz, equipped with up to 4 GByte of RAM and a Linux OS, where at most 3 GByte can be assigned to a single process [Intb]. For this reason current Linux kernels on 32-bit architectures are limited to solve CTMCs consisting of $\approx 1.2 \cdot 10^8$ states. In such a setting the vector holding the elements of the matrix diagonal, and the probability vector already consume ≈ 1.84 GByte of RAM, where at most another ≈ 0.92 GByte must be spent for the iteration vector, depending on the numerical method to be used for computing state probabilities. Consequently the exact number of states of course depends of the size of the symbolic representation of the transition rate matrix.

5.5.1 Comparing ADD and ZDD based numerical solvers

In the above section we already reported that the use of ZDDs may reduce space and time for generating activity-labeled CTMCs for different high-level models by a factor of 2-3, if compared to ADDs. A similar picture can be drawn when it comes to the computation of steady state and transient state probabilities.

(A) Steady state analysis, $b := 0.35$ and $s := 0.35$

FMS

N	JAC with HO *Mt*-DDs				PGS with BHO *Mt*-DDs			
	# iter	t_{iter} in sec.		r_{iter}	# iter	t_{iter} in sec.		r_{iter}
		ADD	ZDD			ADD	ZDD	
6	845	0.1878	0.0945	1.99	569	0.2083	0.0753	2.77
8	1,127	1.5520	0.6445	2.41	737	1.6935	0.5439	3.11
10	1,415	8.7106	4.3969	1.98	892	9.6432	3.8183	2.53

Kanban

N	JOR with HO *Mt*-DDs				PGS with BHO *Mt*-DDs			
	# iter	t_{iter} in sec.		r_{iter}	# iter	t_{iter} in sec.		r_{iter}
		ADD	ZDD			ADD	ZDD	
5	1,977	0.6849	0.3233	2.12	1,542	0.8345	0.2878	2.90
6	2,785	3.2299	1.4929	2.16	2,176	3.8845	1.3681	2.84
7	3,724	10.9477	5.0642	2.16	2,913	15.0764	5.1502	2.93

(B) Transient analysis, $s := 0.7$

FMS

N	Uniform. with HO *Mt*-DDs			
	# step	t_{step} in sec.		r_{step}
		ADD	ZDD	
6	1,508	0.09695	0.0557	1.7406
8	1,864	1.55200	0.8768	1.7701
10	2,217	8.71055	5.3400	1.6312

Kanban

N	Uniform. with HO *Mt*-DDs			
	# step	t_{step} in sec.		r_{step}
		ADD	ZDD	
5	1,157	0.52530	0.3060	1.7167
6	1,157	2.48796	1.4700	1.6925
7	1,157	9.47386	5.5929	1.6939

Table 5.14: ADD and ZDD based solution of CTMCs

Tab. 5.14.A shows the run-time data when computing steady state probabilities for the FMS and Kanban model with a differing scaling parameter N. Here we restrict ourselves to the *JAC* and backward PGS methods, as well as their over-relaxed versions (cf. Sec. 2.2.2, p. 13ff). Both, the sparse as well as the block level were computed by setting scaling factor s and b to 0.35, which is the heuristic reported in [Par02]. From Tab. 5.14.A one may conclude that the employment of ZDDs yields clear runtime advantages, This advantage stems from the maintenance of their compactness under the offset-labeling scheme. As one may recall from Sec. 3.5.3 (p. 66), larger choices of s and b reduce the CPU time consumed per iteration, as well as the number of iterations before reaching steady state. Given now that the linked list for administering the block-entries of the transition matrix, as well as the higher sparseness of ZDDs, allows one in principle to choose larger values for s and b than under the BHO ADD based layout, it is evident that the figures of Tab. 5.14.A are pessimistic concerning the run-time benefits of ZDDs. I.e. if block and sparse level are chosen in such a way that ZDD and ADD based BHO *Mt*-DDs consume almost the same size of memory, the performance of the ZDD based solver can be increased further, so that the differences in time becomes even larger. This issue will be discussed in detail in the next subsection.

Tab. 5.14.B shows the run-time data when computing transient state probabilities, where we employed the uniformization method and the Fox-Glynn method [FG88] for computing the values of the Poisson distribution (cf. Sec. 2.2.2, p. 13). Here also an improvement in CPU time consumption can be observed. However, here ZDDs realize only a speed-up between 1.6 and 1.8. This has to do with the fact that under this setting we decided to set s to 0.7, since memory space was available and doing so speeds up the numerical computation of the solution. As a consequence of this, the sparseness of HO ZDDs is less significant over their ADD based counterparts. Given also the fact that the actual amount of CPU time spent for computing new vector entries (not traversing the *Mt*-DD-structures but computing the

(A) *JAC* based solvers

		$s := 0.35$			$s := 0.7$			Ratios	
N	n_{iter}	t_{iter}	t_g	MByte for HO ZDD	t_{iter}	t_g	MByte for HO ZDD	r_{iter}	r_{mem}
FMS, JAC with HO ZDDs									
6	845	0.13	1.85	0.70	0.05	0.75	4.00	2.45	0.18
8	1,127	0.80	15.04	1.62	0.51	9.50	35.39	1.58	0.05
10	1,415	4.70	110.75	3.33	3.13	73.78	142.15	1.50	0.02
Kanban, JOR with HO ZDDs									
5	1,977	0.35	11.56	0.19	0.32	10.46	7.06	1.11	0.03
6	2,785	1.62	75.28	0.31	1.47	68.08	21.50	1.11	0.01
7	3,724	6.20	384.87	0.48	5.66	351.42	57.18	1.10	0.01

(B) *pGS* based solvers

FMS, PGS with BHO ZDDs

	$b := 0.3, s := 0.35$						$b := 0.5, s := 0.2$					
N	n_{iter}	t_{iter}	t_{sol}	$mem.$	$\|blk\|$	$\#blk$	n_{iter}	t_{iter}	t_{sol}	mem	$\|blk\|$	$\#blk$
6	575	0.14	1.33	0.72	3,136	1,092	472	0.3	2.39	1.56	336	53,101
8	737	0.87	10.63	1.68	11,340	2,937	598	0.93	9.31	5.23	1080	179,834
10	893	5.03	74.85	3.47	32,670	6,578	714	14.96	178.04	14.67	2112	638,267

	$b := 0.5, s := 0.4$						Ratios			
N	n_{iter}	t_{iter}	t_{sol}	mem	$\|blk\|$	$\#blk$	$r_{t_{sol}}$	r_{mem}	$r_{t_{sol}}$	r_{mem}
6	472	0.08	0.60	1.73	336	53,101	2.21	0.42	3.99	0.9
8	598	0.65	6.48	5.19	1,080	179,834	1.64	0.32	1.44	1.01
10	714	3.77	44.86	15.97	2,112	638,267	1.67	0.22	3.97	0.92

Kanban, BPGS with BHO ZDDs

	$b := 0.3, s := 0.35$						$b := 0.5, s := 0.35$					
N	n_{iter}	t_{iter}	t_{sol}	$mem.$	$\|blk\|$	$\#blk$	n_{iter}	t_{iter}	t_{sol}	mem	$\|blk\|$	$\#blk$
5	1,542	0.37	9.40	0.24	21,168	2,408	1,346	0.37	8.36	0.6	1,176	20,041
6	2,176	1.73	62.57	0.41	51,744	4,872	1,900	1.72	54.39	1.28	2,352	47,824
7	2,913	6.62	321.45	0.66	112,320	8,808	2,545	6.67	283.13	2.54	4,320	102,096

	$b := 0.5, s := 0.4$						Ratios			
N	n_{iter}	t_{iter}	t_{sol}	mem	$\|blk\|$	$\#blk$	$r_{t_{sol}}$	r_{mem}	$r_{t_{sol}}$	r_{mem}
5	1,346	0.36	7.99	0.84	1,176	20,041	1.18	0.48	1.05	0.72
6	1,900	1.64	52.02	1.81	2,352	47,824	1.20	0.44	1.05	0.71
7	2,545	6.38	270.70	3.6	4,320	102,096	1.19	0.42	1.05	0.71

Table 5.15: HO ZDD based solution for different sparse and block levels

Poisson probabilities) is also higher than in case of the iterative methods for computing steady state probabilities, it is not surprising that ZDDs realize here smaller speed-ups.

5.5.2 Choice of block and sparse level

As already discussed above, the choice of an adequate sparse-level ($l_{sparse} := (1 - s) \cdot 2n_{\mathcal{V}}$) severely influences the CPU time consumed per iteration of the numerical method applied for computing state probabilities. In contrast the choice of an adequate block level ($l_{blck} := b \cdot 2n_{\mathcal{V}}$) not only interferes with the iteration time, but also influences the number of iterations in case of computing steady state probabilities by the means of the forward or backward PGS method. In order to investigate the impact of the size of s and b on the CPU time and memory consumptions, we analyzed the Kanban - and FMS model in the following settings:

(1) *JAC* based solvers with two different choices for s:

(1.a) scaling factor $s = 0.35$,
(1.b) scaling factor $s = 0.7$

(2) *pGS* based solvers in three different settings:
(2.a) we maintained the sparse level and introduced a block-structure at the top, i.e. $s = 0.35$ and $b = 0.3$.
(2.b) we decreased or maintained the value of the sparse level and increased the block level. I.e. parameter s was set to 0.2 for the FMS model and to 0.35 for the Kanban model, where b was set in both cases to a fixed value of 0.5,
(2.c) we maintained the block level and increased the sparse level, i.e. $s = 0.4$ and $b = 0.5$.

The above settings allowed us to investigate the following effects: (a) the effect of increasing and decreasing the sparse level on the time per iteration and (b) the effect of the number of removed block levels: (i) on the number of iterations and (ii) on the time per iteration.

Tab. 5.15 shows the run-time data when computing steady state probabilities for the FMS and Kanban model in the above settings and different model scaling parameters (N).

Tab. 5.15.A shows the settings investigated for the JAC and JOR method. It indicates the number of iterations (n_{iter}), the time per iteration t_{iter}, the total time for obtaining a numerical solution (t_{sol}), and the memory required for storing the HO ZDD in MByte (col. MByte for HO ZDD). Furthermore it also gives ratios for the iteration times (r_{iter}) and ratios for the memory consumption concerning the HO ZDD (r_{mem}), where everything was normed to the figures obtained for $s := 0.7$.
Tab. 5.15.B shows the settings investigated for the backward *pGS* method. For comparison, the subtable for the different settings also gives the number of iterations (n_{iter}), the time per iteration t_{iter}, the total time for obtaining a numerical solution (t_{sol}), the memory required for storing the BHO ZDD and the iteration vector in MByte (mem), the max. number of elements of the blocks ($|blk|$) and the number of blocks ($\#blk$). Furthermore it also give ratios for the iteration times (r_{iter}) and ratios for the memory consumption concerning the BHO ZDD and iteration vectors (r_{mem}), where the first two settings were normed to the figures obtained for $b := 0.5$ and $s := 0.4$.

Effects on the time per iteration

As one may conclude from Tab. 5.15.A, larger sub-matrices stored in sparse matrix format may decreases the time per iteration. The decrease in run-time obviously depends on the structure of the HO ZDD, since in case of the Kanban model the improvement is only around factor $\frac{1}{1.11}$, whereas in case of the FMS model it varies between $\frac{1}{2.45}$ and $\frac{1}{1.50}$. This effect can be explained as follows: A closer look reveals, that the ZDDs representing the SGs of the different models vary in the number of nodes consumed, where the FMS model requires a larger ZDD (cf. Tab. 5.2, p. 127). Given that the recursive traversal, employed for executing a numerical iteration, stops at each node, it is clear, why the FMS model benefits much more from increasing the number of sparse levels than the Kanban model does.
Tab. 5.15.B shows the settings investigated for the *pGS* based method. If one compares now t_{iter} of Tab. 5.15.A and B, it turns out that the introduction of a block-structuring induces some computational overhead on the time per iteration. However, this effect is only very small and for larger choices of b even becomes smaller (t_{iter} of Kanban model in Tab. 5.15.B as obtained for $s = 0.35$ and $b = 0.3$ or $b = 0.5$). The results of Tab. 5.15.B furthermore indicate that larger sizes of s also clearly reduce the CPU time consumed for each iteration (t_{iter} of the FMS model in Tab. 5.15.B as obtained for $b = 0.5$ and $s = 0.2$ or $s = 0.4$).

Effects on the number of iterations

As one can clearly derive from the data of Tab. 5.15.B a larger choice of b significantly decreases the number of iterations in case of the *pGS* based methods and thus may severely influence the total CPU time consumed for computing steady state probabilities.

		grouped variables				random order				Ratios			
N	n_{iter}	t_{iter}	t_{MGen}	t	sz_M	t_{iter}	t_{MGen}	t	sz_M	r_{iter}	r_{MGen}	r_t	r_{sz}
FMS, with $s := 0.35$													
6	845	0.13	0.92	0.03	0.70	0.19	44.44	0.06	5.76	1.41	48.30	1.80	8.23
8	1,127	0.80	2.66	0.25	1.62	2.93	557.89	1.07	21.32	3.66	209.72	4.26	13.16
10	1,415	4.70	7.43	1.85	3.33	16.82	4,011.27	7.72	56.20	3.58	539.70	4.18	16.86
FMS, with $s := 0.7$													
6	845	0.05	0.85	0.01	4.00	0.05	45.19	0.03	41.59	1.00	53.04	1.96	10.40
8	1,127	0.51	2.92	0.16	35.39	0.46	546.70	0.30	304.69	0.91	186.96	1.86	8.61
10	1,415	3.13	8.74	1.23	142.15	2.86	3,773.43	2.17	1580.18	0.92	431.72	1.76	11.12
Kanban, with $s := 0.35$						ungrouped order							
5	1,977	0.35	0.38	0.19	0.19	0.55	147.33	0.34	7.78	1.57	383.66	1.78	40.89
6	2,785	1.62	1.02	1.25	0.31	2.32	0.87	1.79	17.33	1.43	0.85	1.43	56.01
7	3,724	6.20	2.48	6.42	0.48	8.32	3,383.94	9.54	34.50	1.34	1364.41	1.49	72.07
Kanban, with $s := 0.7$													
5	1,977	0.32	0.49	0.17	7.06	0.36	151.59	0.24	145.26	1.12	308.10	1.36	20.56
6	2,785	1.47	1.30	1.14	21.50	1.61	810.17	1.47	590.51	1.10	621.25	1.30	27.47
7	3,724	5.66	3.34	5.86	57.18	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx

Table 5.16: HO ZDD based solution for different variable orderings

Effects on the consumed memory sizes

Tab. 5.15.A clearly indicates that for larger values of s one has to consider an increase in the size of memory required for storing the transition rate matrix. This is also supported by the FMS model as investigated in the settings of Tab. 5.15.B (cf. *mem* of the FMS model as obtained for $b = 0.5$ and $s = 0.2$ or $s = 0.4$ and of the Kanban model for $b = 0.5$ and $s = 0.35$ or $s = 0.4$).

By comparing the memory sizes for $s = 0.35$ in the settings of Tab. 5.15.A and B, one may conclude that the replacements of upper levels by a block-administering structure minorily increases the memory sizes. However, this effect mainly stems from the circumstance that an increase in b clearly decreases the size of the iteration vector, e.g. for Kanban model $(N = 7, b = 0.5)$ the iteration vector requires only $4,320$ entries rather than the full state vector with $4.1645E7$ entries. This is also supported by the data of the FMS model presented in Tab. 5.15.B. For $s = 0.35$ and an increase of b from 0.3 to 0.5, the number of blocks decreases by over an order, whereas the overall memory size is only increased by factor $2 - 4$.

Thus one may conclude that for large models and adequate choices of s and b, the *pGS* based methods are the methods of choice. The choice of s is hereby crucial, since as the data of the Kanban model indicates minor changes can significantly increases the memory required, but may have only ancillary effect on the time per iteration (cf. last two col. of the ratios in Tab. 5.15.B).

5.5.3 Significance of variable orderings

As already pin-pointed in the previous section, the variable ordering is crucial for the performance of the algorithms operating on *Mt*-DDs. Thus a reordering of the variables not only affects the performance of the activity-local scheme, but also the computation of state probabilities. In order to study this circumstance in greater detail, the Kanban and FMS models were once again analyzed. For computing steady state probabilities for the Kanban model the JOR method was employed, where in case of the FMS model the standard JAC method was used. The experiments were carried out in the following settings:

(1) In the first setting the ordering strategy followed the grouping of variables as usual, where the computation of probabilities was done for different sparse levels ($s = 0.35$ and $s = 0.7$).

(2) In the second setting the random variable ordering for the FMS model was employed once again. In contrast, the variables of the Kanban model were once again ordered by *not*

grouping SVs affected by the same activities together. The computation of probabilities was here also executed for different sparse levels ($s = 0.35$ and $s = 0.7$).

The obtained run-time data is presented in Tab. 5.16. For the evaluation we measured the number of iterations (n_{iter}), the time per iteration (t_{iter}), the time required to generate the required ZDDs and to obtain the HO-ZDD representing the transition rate matrix (t_{MGen}), the total CPU time in hours required for computing steady state probabilities ($t := 3600^{-1}(n_{iter} \cdot t_{iter} + t_{MGen})$), and the size of the HO-ZDD representing the transition rate matrix of the CTMC (sz_M). Besides the measured data, Tab. 5.16 also gives ratios, were everything was normed to the run-time data as taken from the experiments with the grouped SVs. As expected, the grouping strategy turns out to deliver much better run-times and lower memory consumptions. For the *non-grouping* ordering scheme as applied in case of the Kanban model the memory consumption for scaling parameter $N = 7$ and $s = 0.7$ was so large, that the computation of steady state probabilities was not possible.[4] Therefore the comparisons to follow, will make use of the grouping strategy for the vectors of boolean variables encoding the SVs, due to its good results.

5.6 Comparison with other solvers

In this section the ZDD based solvers will be compared to standard sparse matrix technology as well as to the symbolic solvers as implemented within the tool Smart. Since Caspa and Prism employ ADDs for carrying out numerical computations, a comparison to their solvers is not necessary, since the superiority of the ZDD based solvers over their ADD based counterparts was already pointed out in Sec. 5.5.1 (p. 142).

5.6.1 Comparison to the sparse matrix solvers of Möbius

Tab. 5.17 gives the run-time data as obtained during performance analysis of some of the benchmark models. Solving the specified high-level MRMs includes CTMC construction, computation of state probabilities and computation of performance variables. This yields typical measures such as the mean value of a set of SVs or the mean time of the model being in a specific state. For solving high-level MRMs the Möbius tool employs its state-level abstract functional interface (AFI), which makes solver and matrix representation independent of each other [DCKS02]. However, Möbius' sparse matrix solvers considered in this section do not employ this interface, since it induces an additional run-time and memory overhead.

For obtaining steady state solutions the Gauss-Seidel method for the sparse matrix layouts and the *pGS* method for the ZDD based matrix layout were applied. As a consequence, the ZDD based solver had to execute sometimes a clearly increased number of iterations (factor 1.77 up to 6.23). However, as illustrated by Tab. 5.17, this is justified, since the employment of a ZDD based engine within the Möbius modeling environment allows the analysis of models, which were not analyzable under Möbius' conventional schemes for constructing and solving MRM. This shortcoming has to do with the fact that Möbius stores the generated CTMC and its reward information in a non-compact ASCII format on hard disk drive limiting the size of MRMs to be handled ($\sim 5 * 10^6$ states). Not enough, this uncompressed information must be reloaded into RAM and converted into sparse matrix format before the solution process can be initiated (cf. col. *"file reading"* in Tab. 5.17.A, which we did not include in the CPU time consumed for each iteration as given in col. *"each iter."*). In addition to these tool-specific disadvantages, conventional schemes also have the following problems:

(1) the sparse matrix format is hampered by its memory requirements as illustrated in col. *"matrix"* of Tab. 5.17.B.

[4] The vectors for holding the matrix diagonal elements, new state probabilities and the previous ones consume ≈ 976 MByte.

(A) Time consumption of solvers for computing performability measures

	N	Symbolic solver: CPU time in sec. consumed for			Sparse solver: CPU time in sec. consumed for				ratios for	
		t_{MGen}	each iter.	PV calc.	file reading	each iter.	PV calc.		iter. time	PV time
FMS	6	0.29	0.073728	0.11	15.02	0.0562	6.47	1.14	0.76	69.17
	8	0.79	0.61602	0.71	137.58	0.509	56.05	15.57	0.83	100.88
	10	1.64	3.595877	4.15	xxx	xxx	xxx	xxx	xxx	xxx
	12	3.92	16.8645	17.71	xxx	xxx	xxx	xxx	xxx	xxx
Kanban	5	0.51	0.347	0.26	102.86	0.249	30.36	5.6	0.718	138.32
	6	0.58	1.578	1.22	xxx	xxx	xxx	xxx	xxx	xxx
	7	1.17	6.170	4.27	xxx	xxx	xxx	xxx	xxx	xxx
Courier	3	1.78	0.241	0.18	53.36	0.178	29.33	7.16	0.737	202.70
	4	2.92	1.003	0.76	316.53	0.751	118.72	35.11	0.749	202.41
	5	4.54	3.415	2.62	xxx	xxx	xxx	xxx	xxx	xxx
	6	6.61	9.948	7.25	xxx	xxx	xxx	xxx	xxx	xxx
Polling	15	0.03	0.085	0.03	27.21	0.065	8.76	1.5	0.772	341.89
	18	0.06	0.937	0.31	xxx	xxx	xxx	xxx	xxx	xxx
	20	0.08	4.445	1.39	xxx	xxx	xxx	xxx	xxx	xxx
	21	0.09	9.603	2.91	xxx	xxx	xxx	xxx	xxx	xxx

(B) Memory consumption of solvers for computing performability measures

	N	Symbolic solver: MByte of memory consumed for		Sparse solver: MByte of memory consumed for		ratios for	
		overall exec.	matrix rep.	overall exec.	matrix	overall mem.	matrices
FMS	6	21	1.743	80	56.336	3.810	32.332
	8	96	5.248	688	509.032	7.167	96.992
	10	458	16.146	xxx	xxx	xxx	xxx
	12	1876	43.485	xxx	xxx	xxx	xxx
Kanban	5	49	0.835	451	318.778	9.204	381.657
	6	191	1.809	xxx	xxx	xxx	xxx
	7	670	3.600	xxx	xxx	xxx	xxx
Courier	3	60	1.805	323	186.294	5.383	103.210
	4	195	6.581	1190	800.535	6.103	121.640
	5	571	13.449	xxx	xxx	xxx	xxx
	6	1551	24.756	xxx	xxx	xxx	xxx
Polling	15	16	0.501	121	81.562	7.563	162.640
	18	117	1.345	xxx	xxx	xxx	xxx
	20	495	3.129	xxx	xxx	xxx	xxx
	21	1052	3.220	xxx	xxx	xxx	xxx

Table 5.17: Comparison with Möbius' sparse-matrix based solvers

(2) computation of PVs for each state during SG exploration (cf. left figure of col. *"PV calc."* of Tab. 5.17.A), as well as reading the PVs from an ASCII-file or at least allocating a respective PV vector of appropriate size and finally computing the moments and variance of the PV (right figure col. *"PV calc."* of Tab. 5.17.A), induces a significant run-time overhead.

In contrast, the proposed ZDD based scheme generates a symbolic representation of the MRM each time the solver is started (SG generation + conversion of the ZDD representing the transition rate matrix into a BHO ZDD). Hereby the times for generating a ZDD based representation of the CTMC as well as generating ZDD based representations of the reward functions and computing mean and variance of the user-defined PVs, once steady state or transient state probabilities have been computed, is obviously negligible (cf. cols. t_{MGen} and *"PV calc."* of Tab. 5.17.A). Furthermore, the compactness of the (B)HO ZDD based representation speaks to the advantage of our symbolic approach, since it is still superior even though we employed a setting which improves the CPU time consumption per iteration at the disadvantage of space complexity, $b := 0.5$ and $s := 0.4$. This might explain why the ZDD based solvers are not significantly slower than the standard sparse matrix ones. Since the matrix representation under such a choice is still very compact, it is clear that the memory

		Runtime data				Ratios			
						ZDD, s := 0.35		ZDD, s := 0.7	
N	n_{iter}	t_{iter}	t_{MGen}	t_{total} (h)	mem	$r_{t_{total}}$	r_{mem}	$r_{t_{total}}$	r_{mem}
		Sparse matrix, plain							
5	1,977	0.38	201.85	0.26	196.33	1.34	1031.54	1.48	27.79
6	2,786	1.75	1046.81	1.62	1367.14	1.29	4419.09	1.43	63.59
7	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
		Sparse matrix, Kronecker							
5	1,978	2.02	0.004	1.09	0.01	5.65	0.04	6.25	0.001
6	2,786	9.46	0.004	7.2	0.01	5.74	0.04	6.35	0.001
7	3,726	36.20	0.004	36.84	0.02	5.74	0.04	6.29	0.0003
		MxD, Kronecker							
5	1978	4.02	13.65	2.18	9.73	11.28	51.13	12.48	1.38
6	2,786	19.90	64.26	15.16	42.98	12.08	138.94	13.36	2.00
7	3,726	85.43	253.78	87.04	158.90	13.57	331.94	14.86	2.78
		MTMDD, plain							
5	1,977	1.46	135.58	0.82	9.71	4.26	51.04	4.72	1.38
6	2,785	7.33	940.31	5.84	42.96	4.65	138.86	5.14	2.00
7	3,724	30.90	5,154.69	32.84	158.86	5.12	331.87	5.61	2.78

		ZDD, s := 0.35				ZDD, s := 0.7			
N	n_{iter}	t_{iter}	$t_{MGen}{}^5$	t_{total} (h)	mem	t_{iter}	$t_{MGen}{}^4$	t_{total} (h)	mem
5	1,977	0.35	0.38	0.19	0.19	0.32	0.49	0.17	7.06
6	2,785	1.62	1.02	1.25	0.31	1.47	1.30	1.14	21.50
7	3,724	6.20	2.48	6.42	0.48	5.66	3.34	5.86	57.18

Table 5.18: Comparison with Smart's solvers: Run-time data for the Kanban model

space for storing the probability vectors is the limiting factor as the low-level MRMs become larger. This also exhibits another advantage of the ZDD based scheme. The proposed scheme for generating and computing PVs (cf. Sec. 4.5), allows not only the efficient construction of a ZDD based representation and a ZDD based computation of reward functions (col. *"PV calc."* and col. *"PV time."* of Tab. 5.17.A), it also avoids to employ additional vectors for storing the individual reward values of each state as realized by the standard Möbius solver module. This also explains why Möbius' sparse matrix solver modules require significantly more memory for the overall process (cf. columns *"overall exec."* of Tab. 5.17.B).

5.6.2 Comparison with the solvers of Smart

As for the SG generation, the tool Smart also offers a wide range of solvers for computing state probabilities. These solvers are based on standard numerical methods and on different storage techniques for the transition rate matrix. For comparing our ZDD based solvers to the Smart tool we employed the following layouts:

(1) a standard sparse matrix format

(2) *KO* based representation, where the individual sub-matrices are stored in standard sparse matrix format

(3) *KO* based representation stored as matrix diagram (MxD),

(4) a Multi-terminal Multi-valued Decision Diagram (MTMDD).

Tab. 5.18 gives the run-time data as collected when computing steady state probabilities for the Kanban model with different scaling parameters, where as solution method JOR with a

		Runtime data				Ratios: ZDD, s := 0.35		ZDD, s := 0.7	
N	n_{iter}	t_{iter}	t_{MGen}	t_{total} (h)	mem	$r_{t_{total}}$	r_{mem}	$r_{t_{total}}$	r_{mem}
		Sparse matrix, plain							
6	845	0.07	92.02	0.04	34.14	1.35	48.77	3.28	8.54
8	1,127	0.61	1073.82	0.49	311.00	1.95	191.91	3.08	8.79
		Sparse matrix, Kronecker							
6	845	0.24	2.93	0.06	1.09	1.77	1.55	4.30	0.27
		MxD, Kronecker							
6	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
		MTMDD, plain							
6	845	0.26	610.00	0.23	2.05	7.29	2.93	17.69	0.51
		ZDD, h := 0.35				ZDD, h := 0.7			
N	n_{iter}	t_{iter}	t_{MGen}[4]	t_{total} (h)	mem	t_{iter}	t_{MGen}[4]	t_{total} (h)	mem
6	845	0.13	0.92	0.03	0.70	0.05	0.85	0.01	4.00
8	1,127	0.80	2.66	0.25	1.62	0.51	2.92	0.16	35.39
10	1,415	4.70	7.43	1.85	3.33	3.13	8.74	1.23	142.15

Table 5.19: Comparison with Smart's solvers: Run-time data for the FMS model

relative convergence criteria with an accuracy of 10^{-6} was chosen. In the first two columns, Tab. 5.18 records the model scaling parameter (N), and the number of iterations (n_{iter}) required for computing steady state probabilities. In the next four columns Tab. 5.18 gives the time per iteration in seconds (t_{iter}), the CPU time in sec. for generating and constructing the resp. matrix representation (t_{MGen}), the total CPU time in hours consumed for obtaining a solution ($t_{total} := 3600^{-1}(n_{iter} \cdot t_{iter} + t_{MGen})$) and the memory required for storing the transition rate matrix in MBytes (mem). For comparison, the above system was also solved by employing our ZDD based framework. In order to make a fair comparison we once solved the models with a sparse scaling factor of $s := 0.35$ and once for the sparse scaling factor $s := 0.7$. The obtained run-time data is shown at the bottom of Tab. 5.18. The resp. ratios are shown in col. 7-10 of Tab. 5.18, where everything is normed to the figures of the ZDD based framework. As one can see, our ZDD based framework clearly outperforms the solvers of the Smart tool, not only for the solution time, but also for the consumed memory space. Only the *KO* based solver with a standard sparse matrix format is clearly more memory efficient in case of the Kanban model, than our HO ZDD based matrix representations.
From the data presented in Tab. 5.18 one may conclude that *KO* based solvers in combination with symbolic data structures are clearly slower than solvers operating on plain matrices represented by symbolic data structures. This conclusion stems from the fact that our HO ZDD based solvers clearly outperform Smart's symbolic *KO* based solvers (MxD-layout). The above results are even more remarkable, since the Kanban model is highly suited for being analyzed under an approach making use of a *KO* driven composition scheme.

Not surprisingly we had severe problems, when analyzing the FMS model, since we did not find an adequate partitioning and thus analyzed the un-partitioned model. As a consequence the solvers, especially the ones making use of a *KO* based representation, performed poorly and were capable of analyzing the FMS model for small values of N only. The obtained run-time data and ratios are given in Tab. 5.19.

[5] In case of the ZDD based framework this also includes the time for SG generation

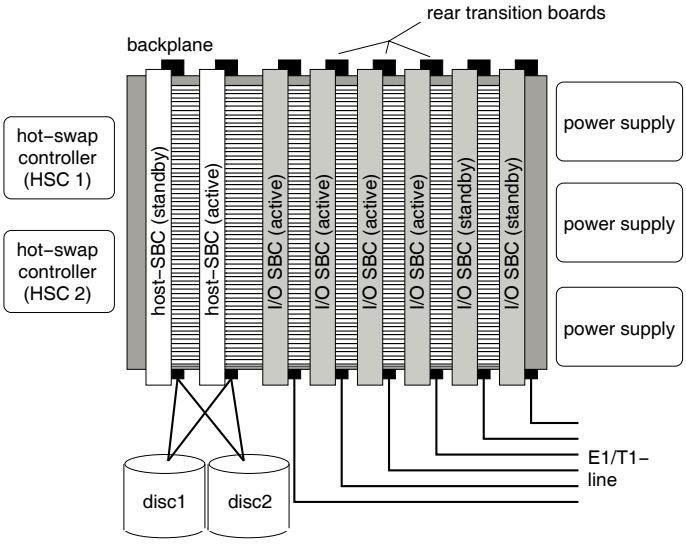


Figure 5.1: Illustration of the adjunct processor (board) system [GLW00]

5.7 Case Study: Telecommunication service system

For demonstrating the applicability of the presented ZDD based framework in practice an adjunct processor system (APS) as employed in the telecommunication service industry was analyzed [GLW00]. However, instead of directly specifying this system as a SAN, it was specified by a set of "user-friendly" input diagrams as accepted by the availability evaluation tool OpenSESAME [WT05]. Since OpenSESAME allows a conversion of its input models into SANs as accepted by the Möbius tool, the APS could be easily made available for being analyzed by our ZDD based framework. I.e. the numerical solution of the low-level MRM, as derived from the generated SAN model specification of the APS, gives one the desired availability measures.

5.7.1 System description

In the digital telephone network APSs are employed for offering specific services to the user or telephone network, e.g. they translate easy-to-remember, location-independent virtual phone numbers into their location-dependent physical equivalent. Since APSs play a crucial role in the network, they must be highly available. Typically, an availability of 99.999% is demanded for such a system, which corresponds to a mean downtime or unavailability of less than 5 minutes per year. From a top-level view, an APS is a series system comprising host units, I/O-units, hot-swap controllers, power supplies, a RAID system and so on (cf. Fig. 5.7.1), where we will evaluate the I/O-subsystem only, as it is the most complex part of the system. The other parts can be evaluated in a similar way which is not done here.

The I/O-unit consists of N sub-units, each comprising a single board computer (SBC) and a so-called rear transition board (RTB). All cabling is connected to the RTB which allows for a quick replacement of the SBC in case of a failure. We assume that $K <= N$ sub-units have to be available at the same time to make the I/O-unit available. Furthermore, we assume that each sub-unit has three states: active, failed, or passive. A passive sub-unit does not perform any work but waits until an active sub-unit fails. After failure detection and localization, the I/O-unit is reconfigured which means that one of the passive sub-units becomes activated (warm standby). The overall time interval between a failure and the completion of the reconfiguration is called the fail-over-time (FOT). A sub-unit fails, if either its SBC or RTB fails. This can happen to both the active and the passive sub-units. From the OpenSESAME model, the SAN as depicted in Fig. 5.2 is derived, where Fig. 5.2 shows only one sixth of the overall SPN structure for the case $N = 6$. It represents sub-unit 1 which is one of the units which are active after system startup. The remaining five sub-units

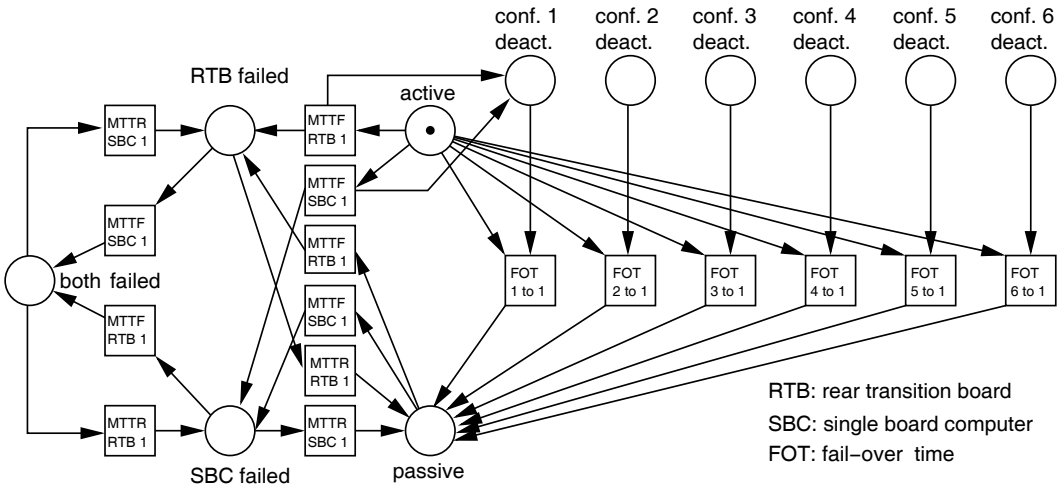


Figure 5.2: Single sub-unit specified as SAN

are modeled with equivalent submodels, however, all submodels share the six SVs `conf. 1-6 deact.`.

The execution rates of the activities of the model are given by the parameters of Tab. 5.20.A, where exponentially distributed time intervals are assumed. Since the parameters of the different sub-units are all identical and so is their structure, the overall model is symmetric, which in principle allows the usage of state lumping, such that a substantially smaller system would have had to be analyzed. But modern architectures of highly reliable systems are often not built from identical sub-components. Thus, all sub-units of a system may have different failure and repair rates, and FOTs, destroying the symmetry of a overall model. Therefore, we did not apply state lumping, so that on the level of the MRM a very large transition system had to be generated and analyzed, illustrating the power of our ZDD based framework.

5.7.2 Model evaluation

The APS was analyzed for varying FOTs and two different settings: (a) In the first setting, we investigated a *"4-out-of-6"* system (4/6), i.e. the system works as long as $K = 4$ from its $N = 6$ sub-units are working. (b) In the second setting a *"6-out-of-8"* system was analyzed (6/8). The two different settings were analyzed on an AMD Opteron 850 2.4 GHz System with 8 or 16 GByte of RAM and a Linux OS [Inta]. For demonstrating the effectiveness of the ZDD based framework we also exported the APS model of OpenSESAME to the GSPN based tool *DSPNexpress* [Lin98], where OpenSESAME also automatically generates the respective model description to be fed into *DSPNexpress*. As convergence criterion of the numerical method computing steady state probabilities a relative convergence of 10^{-6} was chosen. As numerical solution method, the ZDD based framework employed the *pGS*-method, whereas *DSPNexpress* employs the generalized minimal residual method (GMRES). Tab. 5.20.B shows the size of the (un-lumped) low-level MRMs which had to be constructed and solved for obtaining the desired unavailability. Tab. 5.20.B contains the number of system states (*states*), the number of transitions among these system states (*trans*) and the number of transitions explicitly established by the activity-local ZDD based SG generation scheme ($trans_e$).

Tab. 5.20.C shows the memory and CPU-times consumed by *DSPNexpress* and the ZDD based framework when analyzing the high-level MRM as generated by OpenSESAME, where the FOT was set to $6\,min$, which required the largest number of numerical iterations for computing steady state probabilities. Tab. 5.20.C gives the *peak memory* consumption (*mem*), and the CPU time in seconds (*sec*) required for generating the transition rate matrix (t_{MGen}). It also shows the CPU times in *sec* required for executing a single numerical iteration, (t_{iter}), the total number of numerical iterations executed until the convergence criterion is satisfied (*iter*), the CPU time required for constructing and computing the PV

(A) Default parameters of the I/O-unit submodel

parameter	default value	description
N	6 or 8	number of configurations
K	4 or 6	number of initially active configurations
$MTTF - SBC_i$	5E4 hours	mean time to failure of SBC i
$MTTF - RTB_i$	1E5 hours	mean time to failure of RTB i
$MTTR - SBC_i$	1 hour	mean time to repair of SBC i
$MTTR - RTB_i$	1 hour	mean time to repair of RTB i
FOT_{ij}	see Tab. 5.20.D	fail-over-time from configuration i to configuration j

(B) Model specific data

K/N	$states$	$trans$	$trans_e$
4/6	$0.948720\ 10^6$	$1.45607\ 10^7$	240
6/8	$2.61671\ 10^8$	$5.86973\ 10^9$	544

(C) Solution times

K/N	*DSPNexpress*					Symbolic Approach					
	mem	t_{MGen}	t_{iter}	$iter$	t_{total}	mem	t_{MGen}	t_{iter}	$iter$	t_{PV}	t_{total}
4/6	7.4 G	123.01	0.73	12	131.77	36 M	2.04	0.09	46	0.04	6.22
6/8	xxx	xxx	xxx	xxx	xxx	4.1 G	18.14	40.53	49	9.64	2013.75

(D) Model unavailability with respect to FOT

K/N	Varying mean values for the FOT				
	0	$10\,sec$	$1\,min$	$5\,min$	$6\,min$
4/6	$< 5.99 \cdot 10^{-13}$	$1.67 \cdot 10^{-7}$	$9.97 \cdot 10^{-7}$	$4.93 \cdot 10^{-6}$	$5.91 \cdot 10^{-6}$
6/8	$< 1.60 \cdot 10^{-12}$	$2.49 \cdot 10^{-7}$	$1.49 \cdot 10^{-6}$	$7.40 \cdot 10^{-6}$	$8.86 \cdot 10^{-6}$
	prob. for system being unvailable				

Table 5.20: Data as obtained for analyzing the case study

describing the system's unavailability (t_{PV}), and the total time consumed for analyzing the high-level MRM (t_{total}).

Tab. 5.20.C clearly indicates that the ZDD based framework is much more efficient, when it comes to the analysis of high-level MRMs. This means in detail: For the 4/6-model the *DSPNexpress*-tool requires 123.01 *sec*, this includes the time for SG generating, constructing the model's transition rate matrix, stored in a sparse matrix format and the time for generating the reward measures for each state. In contrast the activity-local scheme requires 2.04 *sec* for generating the transition rate matrix, stored as BHO ZDD and another 0.04 *sec* for generating a ZDD based representation of the reward function describing the systems unavailability (incl. the time for computing first and second moment of the reward itself). However, not only for generating the SGs and the PVs the ZDD based framework turns out to be much more efficient than the traditional approach employed by *DSPNexpress*. The extremely high memory consumption of 7.4 GByte of the latter must be due to the hash table holding the reached states, as well as the generator matrix stored in sparse matrix format, since each iteration vector of the GMRES-method only requires $948720 \cdot 16/1024^2 \approx 14.5$ MByte. In contrast, the ZDD based method requires an overall memory consumption of approx. 36 MByte for analyzing the 4/6-model. A closer look reveals that the BHO ZDD based representation of the transition rate matrix requires hereby only 0.9 MByte, whereas the probability vector, the iteration vector and the vector holding the diagonal entries of the generator matrix require approx. 30 MByte of RAM. Thus, in contrast to the sparse-matrix technology as employed by *DSPNexpress*, the bottleneck of the ZDD based framework is given by the different vectors, rather than the storage of the transition rate matrix. Even

in case of the iteration time the ZDD based framework surprisingly exhibits a higher efficency. However, the significantly larger CPU times consumed by *DSPNexpress* for executing a single numerical iteration must be due to the GMRES-method, since plain sparse-matrix-formats usually deliver the lowest times per numerical iteration.
In total, the above features clearly ease the restriction on the numerical analysis of MRMs. Thus it is not surprising that in case of the 6/8-model, the ZDD based framework is still capable of computing the desired unavailability where *DSPNexpress* fails to do so. The failure of the latter is hereby due to the enormous memory requirements of the sparse matrix layout. But even if enough memory were available, it is clear that the CPU time required for generating the SG and generating the reward values for each state explicitly would induce such a high CPU time consumption that an analysis of the 6/8-model by conventional techniques is not feasible anymore.

Tab. 5.20.D shows the computed results. As values for the time delay of activity executions the values of Tab. 5.20.A were employed, where the unavailability for varying mean values of the FOT had to be computed. As it can be seen, the FOT has a significant impact on the unavailability. I.e. in case of assuming a FOT of 0, as it would be the case when applying a standard Fault-Tree based analysis method, one would obtain a very low probability for the system being unavailable. –This case was computed by setting the rate μ_{FOT} of the exponential distribution describing the FOT to 10^9.– However, for a FOT of $\leq 10\,sec$ ($\Rightarrow \mu_{FOT} \leq 360$) one obtains probabilities which are at least 6 orders of magnitude larger than the 0-case.

5.8 Pre-published material

[LS06a] presents the ZDD based activity-local scheme for generating activity-labeled CTMCs. It shows an empirical comparison of ADDs and ZDDs in case of SG generation and presents the run-time data and ratios for demonstrating the effectiveness of the new reachability algorithm. Furthermore it also demonstrates the competitiveness of the activity-local scheme by comparing to the run-time data of the tool Prism, for the FMS and Kanban model. For the Courier and FTMP models [LS06a] also compared the run-time data as produced by the activity-local scheme to the figures published in [DKS03] in order to compare to the MDD based SG generator introduced there.

[LS06b] discussed and presented the ZDD based computation of state probabilities, as well as the ZDD based computation of PVs. In order to demonstrate the usefulness of the approach, [LS06b] presented (a) the data for comparing ADD- and ZDD- based solvers for computing steady and transient state probabilities and (b) the run-time data as obtained when solving MRMs with Möbius conventional "sparse matrix" based approach and our new ZDD based framework.

The APS case study as well as the tool chain for its ZDD based analysis was the subject of [LSW06], where we also presented the comparison with *DSPNexpress*.

6

Conclusion

6.1 Summary

This thesis not only introduces a new type of decision diagram, but also develops a new efficient semi-symbolic approach for the analysis of high-level Markov reward models with very large state graphs. In total this thesis therefore contributes to the alleviation of the state-space explosion problem, as appearing in the context of high-level Markov reward models. In the following we will briefly recapitulate different aspects of the presented work.

6.1.1 Zero-suppressed multi-terminal BDDs

Boolean functions with small satisfaction sets, and where the fulfilling assignments possess many 0-assigned bit positions, may yield large BDD based representations. In such settings *zero-suppressed* BDDs (z-BDDs) [Min93] have shown to be very helpful. In this work, z-BDDs [Min93] are extended to the multi-terminal case, for efficiently representing pseudo-boolean functions and thus the characteristic functions of stochastic transition relations.

Partially shared ZDDs (*p*ZDDs)

For correctly deducing the (pseudo-) boolean function represented by a z-BDD's (or ZDD's) graph, the set of (function) variables of the represented function matters. As a consequence a ZDD-node (and its subgraph) not equipped with a set of function variables does not uniquely represent a pseudo-boolean function, since skipped variables can either belong to non-decisive or zero-suppressed variables. This is problematic, if one operates in a shared BDD-environment, so that uniqueness of nodes is not only a requirement, but an essential pre-condition for efficiency. For solving this problem, this thesis develops the concept of *partially shared* ZDDs (*p*ZDDs), which requires to equip each *p*ZDD-node with a set of function variables, so that (weak) canonicity is guaranteed.

Algorithms for working with *p*ZDDs

This new type of decision diagram requires a re-design of BDD-algorithms [Bry86], where this thesis developed the respective variants, yielding most importantly the *p*`ZApply`, *p*`ZAnd`, *p*`ZRestrict` and *p*`ZAbstract`-algorithm. However, contrary to the theoretical concept these algorithms allow to implement *p*ZDDs within common shared BDD-environment, where nodes are in general not equipped with function variables. This strategy yields the main advantages that one (a) does not need to store the sets of function variables for each node and (b) increases the sharing among the *p*ZDD-graphs, but at the drawback that the semantic of a *p*ZDD-node is not unique anymore. This is significant when testing for equivalence, which in an ordinary shared BDD-environment and standard types of BDDs is reduced to comparing the memory addresses of nodes. Efficient testing for equivalence is an important issue, since while manipulating the graphs of ZDDs, it appears extremely often, not only for testing if the terminal condition for ending a recursion is satisfied, but also when inspecting the pre-computed table for fetching results as computed during previous recursions. For solving this problem, this work follows the following strategy: When operating on the graph of a *p*ZDD Z, one simply passes Z's set of function variables ($\mathcal{V}^{\mathsf{Z}}$) as additional argument to the graph-manipulating algorithm. Each time a test of equivalence is required, one solely needs to compare the memory address of the current *p*ZDD nodes, as well as the sets of function variables. If the respective pairs are identical so are the represented (characteristic) functions.

6.1.2 Activity/Reward-local scheme

If a high-level model's formalism lacks a symbolic semantics, the only way of obtaining a symbolic representation of its underlying Markov Reward Model is to iteratively execute the high-level model specification and generate the symbolic structures on-the-fly until a fixed point is reached. However, if this is done explicitly for each state and each transition, time and peak memory consumption will prevent a successful application of symbolic state graph representation in practice.

Symbolic state graph generation

For efficiently generating the symbolic representation of a CTMC as described by a high-level Markov reward model, the activity-local approach, as developed in this thesis, exploit local information only. I.e. its main idea is the partitioning of the state graph to be generated, into sets of transitions carrying the same label, where each source and target state of a transition is reduced to those positions, on which the transition inducing activity depends on, yielding a transition system for each activity, which we denote as activity-local transition system. The complete transition relation of the high-level model is then obtained by applying a symbolic composition scheme to the activity-local transition systems previously generated. The explicit state graph generation scheme for successively generating the activity-local transition systems, follows hereby a selective breadth-first-search scheme. Such a search strategy is achieved by exploiting a dependency relation on the set of activities and only trying to execute those enabled activities on a newly reached state, which have not already been tested on it. Contrary to standard search schemes, one does not test here the full state descriptor for deciding whether a state has been already explored or not, one only tests those positions of the state descriptor, which are in the dependency set of the respective activity. Furthermore one also tests not all activities, but only those, which depend on the last activity executed in the current state. As a consequence, the explicit state graph generation and encoding of the activity-local scheme is most likely to be partial in practice. If a local fixed point is reached, i.e. if from a given set of states all sequences of dependent activities are extracted explicitly, symbolic composition takes place. The symbolic structure obtained by symbolic composition encodes a set of potential transitions, therefore at this point it is necessary to perform symbolic reachability analysis. In order to speed up the symbolic reachability analysis, we employ an improved symbolic algorithm which is organized as a quasi-depth-first search and works on the activity-local decision diagrams, i.e. on a kind of partitioned transition relation. After symbolic reachability analysis is terminated, the set of reachable states generated so far is obtained. Since this might result in states triggering new (explicit) model behavior, a re-initialization routine is required. If such states exist, this routine may initiate a new round of explicit state graph exploration, encoding, composition and symbolic reachability analysis. Several rounds may be required until a global fixed point is reached and a complete representation of the high-level model underlying CTMC is constructed.

Generating symbolic representation of reward functions

For describing performability measures of the system under study modeler are enabled to specify performance variables on the level of the high-level model description. Traditionally one computes the state and activity dependent values of performance variables, i.e. the values of their rate and impulse rewards, while generating the low-level representation of the model. However, with the activity-local scheme only a fraction of states is visited explicitly during state graph generation. Given that a reward returning function may be of arbitrary complexity, but solely depends on a subset of state variables, it seems therefore reasonable, to generate their symbolic representation, once the symbolic representation of the overall state graph generated. In order to explicitly process as few states as possible, this works once again exploits local information only. I.e. after computing the reward value of a state, only those positions within the state descriptor are symbolically encoded, the currently processed

reward function depends on. This strategies defines once again an equivalence relation on the set of system states concerning a specific reward function, so that the number of states that need to be explicitly processed, can be reduced significantly. By simply aggregating the symbolically represented reward functions according to the user-defined specification, one obtains symbolic representations of the performance variables. Now the probability distribution on the set of system states must be computed, since state probabilities in combination with symbolically represented performance variables can be combined, so that the performability measures of the system under study are obtained.

6.1.3 Computation of state probabilities

For computing state probabilities, this work presents a ZDD based variant of the hybrid solution method, as already known for other symbolic data structures. In this approach, the generator matrix is represented by a decision diagram and the iteration vectors are stored as ordinary arrays of doubles. If n Boolean variables are used for state encoding, there are 2^n potential states of which only a small fraction may be reachable. Allocating entries for unreachable states in the vectors would be a waste of memory space and would severely restrict the applicability of the systems to be analyzed. E.g. for storing the probabilities of about 134 million states one already requires 1 GByte of RAM. Therefore a dense enumeration scheme for the reachable states has to be implemented. This is achieved via the concept of offset-labeling. The offset-labeling of nodes allows to compute row and column index in the dense enumeration scheme for each matrix element, while traversing the symbolic structure representing the matrix.
The space efficiency of symbolic matrix representation comes at the cost of computational overhead, caused by the recursive traversal of the symbolic structure. For that reason one replaces the lower levels of the symbolic structure by explicit sparse matrix representations, which works particularly well for block-structured matrices. For exploiting the good convergence of the forward and backward Gauss-Seidel method and its over-relaxed variants it is furthermore necessary to access the matrix elements in a row or column-wise manner. As a compromise, we implemented the so-called pseudo Gauss-Seidel iteration scheme as known for Algebraic Decision Diagram based matrix representations. This schemes partitions the matrix into blocks, which are processed in an order fashion, where within each block matrix entries are accessed in arbitrary order. Overall the above approaches, yield a memory structure in which some levels from the bottom and may be also some levels from the top of the symbolic matrix representation have been replaced by sparse matrix structures. By applying an iterative solution method one is then enabled to compute the state probabilities, which are stored as entries of an array.

6.1.4 Computing performability measures

Given symbolic representations of the performance variables and the state probabilities, one is enabled to compute the performability measures for the system under study. For doing this, the thesis introduces a new algorithm, which computes moments of the symbolically represented performance variables via graph-traversion.

6.2 Benefits of *p*ZDDs and the activity/reward-local scheme

The advantages of the presented concept of *p*ZDDs can be summarized as follows:

- The concept of *p*ZDDs and its introduced algorithms, allows the efficient allocation and manipulation of *z*-BDDs and their multi-terminal extensions within a common BDD-Package, even though the DDs may not have equal sets of function variables. To the best of our knowledge this aspect was not subject of previous works.
- In the context of symbolic representation of set of states as derived from high-level model descriptions, ZDDs showed that they are often the most memory-efficient data type. This advantage can also often be observed, when it comes to the representation of stochastic transition systems.
- From the measurements one can conclude that ZDDs perform better then their ADD based counterparts, if the generation of symbolic SG representations and the hybrid solution method for solving sets of differential and linear equations is used. Based on their compactness one may convert larger fractions of the ZDD into sparse matrix layouts reducing the solution time even more and/or reduce the size of the additional iteration vector in case of the pseudo Gauss-Seidel method and thus analyze larger systems.

The benefits of the new semi-symbolic, compositional and submodel-interdependent technique for generating symbolic representations of high-level model's underlying transition systems, presented in this thesis, can be summarized as follows:

- In general, only a small fraction of the transitions of the Markov chain needs to be generated and encoded explicitly, whereas the bulk of the transitions is generated during symbolic composition.
- The approach is compositional, which contrary to monolithic approaches keeps not only the run-time at a moderate level, but also the peak memory consumption.
- The reachability analysis of the high-level model is carried out efficiently at the level of the symbolic data structure. where this work presents a new quasi-depth-first-search scheme with a convincing efficiency.
- For the generation of symbolic representation of reward functions explicit computation can also be limited to a small fraction of states. This is much more efficient in comparison to the traditional approaches, where the reward values for each state is explicitly computed.
- This works suggests a symbolic representation of user-defined performance variables and a computation of their moments via a new graph-traversing algorithm, which reduces memory and run-time, if compared to standard approaches.
- Since the activity/reward-local approach depends on an explicit execution of activity and reward functions, the approach is not limited to a certain high-level model description technique.
- The model is partitioned automatically at the level of the individual activities, i.e. a user-defined partitioning as for other symbolic techniques is not necessary.
- The scheme does not require any particular model structure, i.e. contrary to other well known approaches, the method is not restricted to models that satisfy a *KO compliant* structure.

6.3 Future work

Future steps for improving the approach presented in this thesis, but also symbolic techniques in the context of performability analysis in general, have different aspects, which will now be covered individually.

Improving the state graph generation procedure

As illustrated by the TQN-model (Sec. 5.2.1, p. 123), worst case scenarios for the activity/reward-local scheme exists. Also the performance of the scheme in case of *KO compliant* models, like the Kanban-model, seems to be improvable. A closer look at the high-level models reveal, that some activities contained in the high-level model specification are extensively executed. On the other hand it is known that the state graph of these models can be efficiently constructed when a *Sync* driven decomposition strategy is applied (cf. Sec. 2.5.3, p. 28). I.e. in case the user defines such a decomposition (or the overall model was constructed in a compositional style), the activity/reward-local procedure can be employed for efficiently generating the local state graphs of the different model partitions. A symbolic representation of the overall model can then be obtained by applying the *KO* driven composition scheme of Eq. 2.19 (p. 28), the required symbolic realization can be found [Sie02].

Symbolic execution of activities

The activity/reward-local scheme is a semi-symbolic and compositional approach for generating a symbolic representation of a MRM. Thus it requires the use of a symbolic composition and symbolic reachability scheme. The respective schemes are hereby carried out separately by making extensively use of DD-manipulating algorithms. A good idea seems therefore to integrate composition and one-step reachability computation into a single DD based algorithm. This algorithm takes a symbolic representation of an activity-local transition system and the set of visited states as input and is repetitively executed until a fixed point is reached and a representation of the set of reachable states is generated. In particular such a procedure would not require to apply a composition scheme and would combine different DD-operations in a single algorithm, which in total may yield an improved caching behavior concerning the pre-computed table.

Dynamically changing structures of high-level models

The activity/reward-local scheme as represented in this thesis was designed on the basis of statically structured high-level models. However, an interesting class of high-level model specification techniques, which also allow a mapping of high-level models to finite state graphs, incorporates mechanisms for process instantiation and process deletion during runtime, e.g. [CB06]. The activity/reward-local scheme seems to be extendable, so that it is applicable in such a context. In combination with ZDDs, where the introduction of new state variables does not causes any significant overhead, this seems very promising.

Parallelization of the activity/reward-local scheme

In [Wei05] we gathered first experience with a parallelization of the activity/reward-local scheme. It turned out that the mutual exclusion of write-operations on the ZDDs slows down the overall process of symbolic state graph generation. However, the implementation of a thread-pool allowed the author of [Wei05] to executed also other steps of the activity/reward-local scheme in parallel, which seems to be more promising, especially for models, which require extensive state graph exploration. However, since symbolic reachability analysis is still the main source of CPU-time consumption, a deeper investigation of how-to parallelize symbolic algorithms could bring improvements.

Improving numerical analysis

As pointed out in Sec. 5.5 the major bottle-neck of symbolic approaches for the state graph based analysis of high-level MRMs is the huge number of states, since the probabilities of the latter must be computed individually. For improving the current methods a three-fold strategy can be followed:

(1) *Symbolic algorithms for realizing bisimulation*: In Sec. 4.6.1 (p. 103ff) a symbolic algorithm for computing a reduced bisimilar CTMC was presented. However, this approach was limited to the case that user-defined symmetries within the high-level model are defined. In case a model specification lacks such an explicit definition of symmetries more general algorithms can be employed. However, these algorithms, those symbolic variants are introduced in [Sie02] are known to be not very efficient. In [DKS05] an approach for detecting bisimilar structures on the level of symbolic state graph representations was introduced. The suggested procedure seems quite promising, since it detects submodel imposed symmetries by investigating the symbolic representation of the state graph. A procedure for detecting submodel imposed symmetries in the context of the activity/reward-local scheme is straight forward: One solely needs to compose the activity-local transition systems for each submodel separately and compare them. Such a comparison could either be realized by a respective symbolic algorithm, or a relabeling of variables the submodel-local transition systems are defined on, and a subsequent test of equivalence. As result one would immediately know if the submodels referring to the same specification also exhibit the same (timed) behavior. As major advantage of such a procedure, the overall model would not need to be composed by employing the *Rep*-operator explicitly and thus employ the *Join*-operator in a more flexible way, so that a high-level model can have an arbitrary compositional (graph) structure, but one is still capable of exploiting submodel-imposed symmetries [Oba98]. Going a step further, it seems also very promising to study the effect of symmetries directly on the symbolic representation of the state graph. Efficiently identifying such symmetries and exploiting them, would yield a minimal bisimulation relation, whereas the exploitation of submodel-imposed symmetries not necessarily does.

(2) *Approximative solution methods:* The multi-level approach [HL94] is known to be also efficiently applicable, in situations, where the generator matrices of the aggregated systems are represented symbolically and/or implicitly. Given that extensive manipulations of large symbolic data structures are computationally expensive, it is clear that the multi-level approach in the context of symbolic representations is only efficiently applicable as long as the transition rate matrices of the different aggregation levels are not generated each time the respective aggregation level is accessed [Buc06]. However, the multi-level method still needs to allocate a probability vector, its size is bounded by the number of states. Consequently the applicability of approaches making use of the multi-level method is still limited in practice. –In [BG05] the authors presented an approach, which seems to be inspired by the multi-level method, but starting with a state graph which is already aggregated, so that the number of state probabilities to be computed is clearly reduced. An approximated result is then obtained, by varying the aggregation of states, computing state probabilities for each of the aggregated systems and manipulating the transition rates between the aggregated states. This procedure is repeated until a fixed-point is reached. It would be interesting, to study the question, if such a procedure works for any high-level model description technique and for arbitrarily aggregated system states and how such a procedure can be efficiently executed on the level of symbolically represented state graphs.

(3) *Parallelization*: The authors of [BH01] introduce a parallelized numerical solver, which makes use of standard sparse matrix structures. It is straight-forward to make here also use of symbolic structures, where the pseudo Gauss-Seidel method seems very promising for being parallelized.

A

Appendix: Mathematical Background

It is assumed that the reader is familiar with the concept of boolean algebra. In the following paragraphs we will briefly give some basic definitions and operations as employed in the concourse of the thesis.

A.1 Boolean functions

A n-ary boolean function $f(\mathsf{v}_1, \ldots, \mathsf{v}_n)$ is a mapping $\mathbb{B}^n \to \mathbb{B}$, where $\mathbb{B} := \{0, 1\}$. The set of variables $(\mathsf{v}_1, \ldots, \mathsf{v}_n)$ employed in function f is denoted as set of function variables, where we will use the notation $(\mathcal{V}^f)$.

0-maintaining operator

A n-ary operator $\mathtt{op}$ is *0-maintaining iff* $0\,\mathtt{op}\,\ldots\,\mathtt{op}\,0 = 0$ holds.

Decisive variables

A variable v_i is decisive for a boolean function if at least for an assignment to the function variables of f holds that $f(\mathsf{v}_1, \ldots, \mathsf{v}_{i-1}, 0, \mathsf{v}_{i+1}, \ldots, \mathsf{v}_n) \neq f(\mathsf{v}_1, \ldots, \mathsf{v}_{i-1}, 1, \mathsf{v}_{i+1}, \ldots, \mathsf{v}_n)$.

Minterm and Monomial

For a boolean n-ary function f a product term is a conjunction of a set of literals from its set of variables, where a literal is either the boolean variable itself or its negation. A product term in which each of the n (function) variables appears at most once is called a monomial. A minterm is a product term, where each of the n (function) variables appear exactly once.

Sum-of-Products

A Boolean function expressed as a disjunction of monomials is commonly known as the sum of products (SOP). I.e. in this representation product terms may or may not cover the same minterm. If each product term of the SOP is a minterm one speaks of the canonical SOP or canonical disjunctive normal form (CDNF). In case no two product terms cover the same minterm, one speaks of a disjoint sum of products (DSOP). Thus the canonical disjunctive normal form is the special case of a DSOP.

Canonical representations

Two boolean functions are *equivalent*, if their function values coincident for all inputs to their function variables. A representation of a boolean function is called *canonical* if each function f has exactly one representation of this type. The representation is denoted *universal* if each boolean function possesses a representation of this type. A representation is denoted as *strongly canonical*, if two equivalent functions share the same representation. I.e. functions with identical sets of decisive variables have always the same representation, no matter what their set of function variables are. If for identical sets of decisive variables and different sets of function variables the representation of two equivalent function is not the same, we denote them as *weakly canonical*. In this sense the canonical disjunctive normal form is a weak canonical representation.

Expansion
A boolean function defined on a set of boolean variables $\mathcal{V}$ can be expanded with respect to a variable v_i by replacing each of its occurrences with the constants 1 and 0. The resulting functions, which are independent of the v_i, need than to be combined via disjunction in order to substitute f.

Definition A.1: Expansion of Boolean functions

Let $f : \mathbb{B}^n \to \mathbb{B}$ be a boolean function and let $\mathcal{V}^f$ be the set of f's function variables. For all $v_i \in \mathcal{V}^f$ it holds:

$$f := v_i f(v_1, \ldots, v_{i-1}, 1, v_{i+1}, \ldots, v_n) + \neg \mathsf{v}_i f(v_1, \ldots, v_{i-1}, 0, v_{i+1}, \ldots, v_n)$$

The expansion can recursively applied until all n variables are made constant. Each of the terms to be connected via disjunction, gives then a line-entry of the truth table for a specific assignment $\vec{b}$ with respect to the (ordered) function variables $\vec{\mathsf{v}}$. In case one discards all terms where f is evaluated to 0, one ends up with the CDNF for function f.

The above defined expansion of boolean functions is also commonly denoted as *Shannon expansion*. In 1938 Shannon introduced this theorem in the context of switching functions, so that their mapping to a boolean algebra was achieved [Sha00].

Co-factors and sub-functions
If one expands only one variable, e.g. v_i one ends up with the two co-factors of f with respect to v_i. In case v_i was replaced by the constant 1 one speaks than of the positive - or one co-factor ($f|_{\mathsf{v}_i:=1} := f(\mathsf{v}_1, \mathsf{v}_{i-1}, 1, \mathsf{v}_{i+1}, \ldots \mathsf{v}_{n_\mathcal{V}})$), where in case v_i was replaced by the constant 0 one speaks of the negative - or zero co-factor ($f|_{\mathsf{v}_i:=0} := f(\mathsf{v}_1, \mathsf{v}_{i-1}, 0, \mathsf{v}_{i+1}, \ldots, \mathsf{v}_{n_\mathcal{V}})$). The expansion can be applied for an arbitrary subset of $\mathcal{V}$, e.g. $\mathcal{V}'$, so that the notation $f|_{\vec{\mathsf{v}}':=\vec{b}}$ refers to the co-factor of f for the variables of $\mathcal{V}'$ and with respect to the assignment $\vec{b}$.

A.2 Pseudo-boolean functions

A n-ary pseudo-boolean function $f(\mathsf{v}_1, \ldots, \mathsf{v}_n)$ is a mapping $\mathbb{B}^n \to \mathbb{D}$, where $\mathbb{D}$ is a finite set, e.g. $\mathbb{D} \subset \mathbb{R}$. All results on boolean functions remain valid for pseudo-boolean functions.

A.3 Kronecker operators

(1) The Kronecker product (KP) $\oplus : \mathbb{R}^{n,m} \times \mathbb{R}^{k,l} \to \mathbb{R}^{nk,ml}$ is defined as follows:

$$C := A \otimes B = \begin{bmatrix} a_{1,1}B & a_{1,2}B & \cdots & a_{1,m}B \\ a_{2,1}B & \ddots & \vdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1}B & a_{1,2}B & \cdots & a_{n,m}B \end{bmatrix}$$

As one can see under the KP one combines the matrices A and B in an element-to-block-wise manner, where each block is equal to matrix B, multiplied with matrix entry $a_{i,j}$. This yields:

$$C((i_1, i_2), (j_1, j_2)) = A(i_1, j_1)B(i_2, j_2) \tag{A.1}$$

where $C((i_1, i_2), (j_1, j_2))$ addresses block (i_1, j_1) and element (i_2, j_2) within this block. The KP can be extended to sets of matrices inductively as follows:

$$C := \bigotimes_{i:=0}^{n} A_i = \left(\bigotimes_{i:=0}^{n-1} A_i \right) \otimes A_n = \prod_{i:=0}^{n} 1_{d_i^1} \otimes A_i \otimes 1_{d_i^2}$$

where $\mathbb{1}_{d_i^j}$ is an identity matrices of dimension $d_i^1 := \prod_{x:=1}^{i-1} \dim(A_x)$, $d_i^2 := \prod_{x:=i+1}^{n} \dim(A_x)$ respectively.

(2) The Kronecker sum (KS) $\oplus : \mathbb{R}^{n,n} \times \mathbb{R}^{k,k} \to \mathbb{R}^{nk,nk}$ is defined as follows:

$$C := A \oplus B = A \otimes \mathbb{1}^{k,k} + \mathbb{1}^{n,n} \otimes B$$

This definition can be extended to sets of matrices inductively as follows:

$$C := \bigoplus_{i:=0}^{n} A_i = \left(\bigoplus_{i:=0}^{n-1} A_i \right) \oplus A_n = \sum_{i:=0}^{n} \mathbb{1}_{d_i^1} \otimes A_i \otimes \mathbb{1}_{d_i^2}$$

where $\mathbb{1}_{d_i^j}$ is once again an identity matrices of appropriate dimension.

A.4 Notation of modus ponens

In the style of SOS-semantics of process algebras we will employ the following notation:

$$\frac{\text{premise}_1; \ldots ; \text{premise}_n}{\text{conclusion}} \quad \text{side condition}_1 \ldots, \text{side condition}_m$$

The meaning of this notation is as follows: if all side conditions are true the conclusion can be drawn, given that the premises hold. In case the set of premises and the set side conditions is empty, the conclusion can always be drawn.

A.5 Pseudo-code and related notation

The pseudo-code presented in this thesis will follow the style of the C-programming language.

(1) Operators of propositional logic:
In case of logic-connectors applied on predicates we will employ the symbols && , for representing $\wedge$ and the symbol $\|$ for representing $\vee$.

(2) Test of inclusion:
The operation $n, m \in \mathcal{K}_T$ gives true, if $\{n, m\} \subseteq \mathcal{K}_T$ holds, otherwise false.

(3) Test of equality:
If two variables are equal is tested with the operator $=$.

(4) Assignment:
A value is assigned to a variable by employing the operator $:=$.

(5) Set manipulation:
- Insertion: The operation $B \longleftarrow a$ is equal to $B := B \cup \{a\}$.
- Extraction: $a \longleftarrow B$ is equal to $B := B \setminus \{a\}$, where the entity a is now accessible in the context of the respective block executing the extraction. $x \xleftarrow{a} B$ refers to the operation $B := B \setminus \{a\}$ and $x := a$.

B

Appendix: Algorithms for BDDs and derivatives

```
Satisfy((b_1, ..., b_{n_ν}), n, v⃗)
(0)   i := 1;
(1)   node k := n;
(2)   WHILE k ∉ K_T DO;
(3)      WHILE v⃗[i] <_π var(k) DO i++; END
(4)      IF b_i = 1 THEN k := then(k);
(5)      ELSE IF b_i = 0 THEN k := else(k);
(6)   END
(7)   RETURN value(k);
```

Algorithm B.1: The `Satisfy`-algorithm for BDTs and BDDs

```
Satisfy((b_1, ..., b_{n_ν}), n, v⃗)
(0)   i := 1;
(1)   node k := n;
(2)   WHILE k ∉ K_T DO
(3)      FOR v⃗[i] <_π var(n) DO
(4)         IF b_i = 0 THEN i++;
(5)         ELSE RETURN 0;
(6)      END
(7)      IF b_i = 1 THEN k := then(k);
(8)      ELSE IF b_i = 0 THEN k := else(k);
(9)   END
(10)  RETURN value(k);
```

Algorithm B.2: The `Satisfy`-algorithm for z-BDDs

```
Satisfy((b_1, ..., b_{n_ν}), n, v⃗)
(0)   i := 1;
(1)   node k := n;
(2)   WHILE k ∉ K_T DO
(3)      IF b_i = 1 THEN
(3)         IF var(k) <_π v⃗[i] THEN RETURN 0;
(3)         ELSE k := then(k);
(4)      ELSE IF b_i = 0 && var(k) ≤_π v⃗[i] THEN RETURN 0;
(5)      i++
(6)   END
(7)   RETURN value(k);
```

Algorithm B.3: The `Satisfy`-algorithm for pZDDs

C

Appendix: Algorithms for handling models with immediate activities

ExploreStates()
(1) WHILE ($StateBuffer \neq empty$) DO
(2) $(\vec{s}^{\,l}, \mathcal{F}^{D_l}_{\vec{s}^{\,l}}) \longleftarrow StateBuffer$;
/* *Processing dependent immediate activities* */
(3) FOR $k \in \mathcal{A}ct^i \cap \mathcal{F}^{D_l}_{\vec{s}^{\,l}}$ DO
(4) $\vec{s}^{\,lk} := \delta_k(\vec{s}^{\,l})$;
(5) $\lambda := \Pi(\vec{s}^{\,l}, k, \vec{s}^{\,lk})$;
(6) IF $\vec{s}^{\,l} \neq \vec{s}^{\,lk}$ THEN $TransBuffer \longleftarrow (\vec{s}^{\,l}, k, \lambda, \vec{s}^{\,lk})$;
(7) END
/* *Processing dependent Markovian activities* */
(8) FOR $k \in \mathcal{A}ct^m \cap \mathcal{F}^{D_l}_{\vec{s}^{\,l}}$; DO
(9) $\vec{s}^{\,lk} := \delta_k(\vec{s}^{\,l})$;
(10) $\lambda := \Lambda(\vec{s}^{\,l}, k, \vec{s}^{\,lk})$;
(11) IF $\vec{s}^{\,l} \neq \vec{s}^{\,lk}$ THEN $TransBuffer \longleftarrow (\vec{s}^{\,l}, k, \lambda, \vec{s}^{\,lk})$;
(12) END
(13) END
(14) RETURN ;

Algorithm C.1: Explicit SG exploration in the presence of immediate activities

`EncodeTransitions`()
(1) WHILE ($TransBuffer \neq empty$) DO;
(2) $(\vec{s}, l, \lambda, \vec{s}^{\,l}) \longleftarrow TransBuffer$;
(3) $\mathcal{F}^{D_l}_{\vec{s}^{\,l}} := \emptyset$;
/* *Testing all immediate activities for enabledness* */
(4) FOR $\forall k \in \mathcal{A}ct^i : \vec{s}^{\,l}_{D_k} \notin \mathsf{E}_k$ DO
(5) IF $\vec{s}^{\,l}$ $[> k$ THEN $\mathcal{F}^{D_l}_{\vec{s}^{\,l}} := \mathcal{F}^{D_l}_{\vec{s}^{\,l}} \cup \{k\}$;
(6) $\mathsf{E}_k := \mathsf{E}_k + \mathtt{Encode}(\mathcal{E}(\vec{s}^{\,l}_{D_k}), k, \mathcal{V}_{\mathsf{s}}^{\mathsf{D}_k})$
(7) END
/* *Testing only l's dependent Markovian activities for enabledness* */
(8) FOR $k \in \mathcal{A}^{D_l} \cap \mathcal{A}ct^m$ DO;
(9) IF $\vec{s}^{\,l}_{D_k} \notin \mathsf{E}_k \wedge \vec{s}^{\,l}$ $[> k$ THEN $\mathcal{F}^{D_l}_{\vec{s}^{\,l}} := \mathcal{F}^{D_l}_{\vec{s}^{\,l}} \cup \{k\}$;
(10) $\mathsf{E}_k := \mathsf{E}_k + \mathtt{Encode}(\mathcal{E}(\vec{s}^{\,l}_{D_k}), k, \mathcal{V}_{\mathsf{s}}^{\mathsf{D}_k})$
(11) END
/* *Insert state and activity list into buffer, and transition into act-local symbolic structure* */
(12) IF $\mathcal{F}^{D_l}_{\vec{s}^{\,l}} \neq \emptyset$ THEN $StateBuffer \longleftarrow (\vec{s}^{\,l}, \mathcal{F}^{D_l}_{\vec{s}^{\,l}})$;
(13) $\mathsf{Z}_l := \mathsf{Z}_l + \mathtt{Encode}(\mathcal{E}(\vec{s}_{D_l}, \vec{s}^{\,l}_{D_l}), \lambda, \mathcal{V}^{\mathsf{D}_k})$;
(14) END
(15) RETURN ;

Algorithm C.2: Encoding state-to-state-transitions and testing for further expolration

ReachabilityAnalysis()
(0) $\mathsf{Z}_M := \sum_{l \in \mathcal{A}ct^m} \mathsf{Z}_l \times \mathbb{1}_l$;
(1) $\mathsf{Z}_P := \sum_{l \in \mathcal{A}ct^i} \mathsf{Z}_l \times \mathbb{1}_l$;
(2) $\mathsf{Z}_R := \mathtt{Encode}(\mathcal{E}(\vec{s}^{\,\epsilon}), 1, \mathcal{V}_\mathsf{t})$;
(3) $\mathsf{Z}_U := \mathtt{Encode}(\mathcal{E}(\vec{s}^{\,\epsilon}), 1, \mathcal{V}_\mathsf{s})$;
(4) $\mathsf{Z}_{Van}, \mathsf{Z}_{Tan} := \emptyset$;
(5) DO
(6) $\quad \mathsf{Z}_{tmp} := \mathsf{Z}_P \cdot \mathsf{Z}_U$;
(7) $\quad \mathsf{Z}_{trans} := \mathsf{Z}_{trans} + \mathsf{Z}_{tmp}$;
(8) $\quad \mathsf{Z}_{new} := p\mathtt{ZAbstract}(\mathsf{Z}_{tmp}, \mathcal{V}_\mathsf{t}, +)$;
(9) $\quad \mathsf{Z}_{Van} := \mathsf{Z}_{Van} + \mathsf{Z}_{new}$;
(10) $\quad \mathsf{Z}_U := \mathsf{Z}_U \setminus \mathsf{Z}_{new}$;
(11) $\quad \mathsf{Z}_{new} := p\mathtt{ZAbstract}(\mathsf{Z}_{tmp}, \mathcal{V}_\mathsf{s}, +)$;
(12) $\quad \mathsf{Z}_{tmp} := \mathsf{Z}_U \cdot \mathsf{Z}_M$;
(13) $\quad \mathsf{Z}_{trans} := \mathsf{Z}_{trans} + \mathsf{Z}_{tmp}$;
(14) $\quad \mathsf{Z}_U := p\mathtt{ZAbstract}(\mathsf{Z}_{tmp}, \mathcal{V}_\mathsf{t}, +) \setminus \mathsf{Z}_R$;
(15) $\quad \mathsf{Z}_{Tan} := \mathsf{Z}_{Tan} + \mathsf{Z}_U$;
(16) $\quad \mathsf{Z}_U := p\mathtt{ZAbstract}(\mathsf{Z}_{tmp}, \mathcal{V}_\mathsf{s}, +) \setminus \mathsf{Z}_R$;
(17) $\quad \mathsf{Z}_R := \mathsf{Z}_R + \mathsf{Z}_U$;
(18) $\quad \mathsf{Z}_U := \mathsf{Z}_U\{\mathcal{V}_\mathsf{s} \leftarrow \mathcal{V}_\mathsf{t}\}$;
(19) $\quad \mathsf{Z}_U := \mathsf{Z}_U + \mathsf{Z}_{new}\{\mathcal{V}_\mathsf{s} \leftarrow \mathcal{V}_\mathsf{t}\}$;
(20) END UNTIL $\mathsf{Z}_U = \emptyset$
(21) RETURN Z_R;

(A) Breadth-first-search scheme

ReachabilityAnalysis()
(0) $\mathsf{Z}_U, \mathsf{Z}_{Van}, \mathsf{Z}_{Tan} := \emptyset$;
(1) $\mathsf{Z}_R := \mathtt{Encode}(\mathcal{E}(\vec{s}^{\,\epsilon}), 1, \mathcal{V}_\mathsf{s})$;
(2) FOR $k \in \mathcal{A}ct$ DO $\widetilde{\mathsf{Z}_k} := \mathsf{Z}_k \times \mathbb{1}_k$; END
(3) DO
(4) $\quad$ WHILE $\mathsf{Z}_U \neq \emptyset$ DO
(5) $\quad\quad$ FOR $k \in \mathcal{A}ct^i$ DO
(6) $\quad\quad\quad \mathsf{Z}_{trans} := \widetilde{\mathsf{Z}_k} \cdot \mathsf{Z}_U$;
(7) $\quad\quad\quad \mathsf{Z}_{new} := p\mathtt{ZAbstract}(\mathsf{Z}_{trans}, \mathcal{V}_\mathsf{s}, +) \setminus Z_R$;
(8) $\quad\quad\quad \mathsf{Z}_U := \mathsf{Z}_U + \mathsf{Z}_{new}\{\mathcal{V}_\mathsf{s} \leftarrow \mathcal{V}_\mathsf{t}\}$;
(9) $\quad\quad\quad \mathsf{Z}_{new} := p\mathtt{ZAbstract}(\mathsf{Z}_{trans}, \mathcal{V}_\mathsf{t}, +) \setminus Z_R$;
(10) $\quad\quad\quad \mathsf{Z}_{Van} := \mathsf{Z}_{Van} + \mathsf{Z}_{new}$;
(11) $\quad\quad$ END
(12) $\quad\quad \mathsf{Z}_U := \mathsf{Z}_U \setminus \mathsf{Z}_R$;
(13) $\quad\quad \mathsf{Z}_R := \mathsf{Z}_R + \mathsf{Z}_U$;
(14) $\quad$ END
(15) $\quad \mathsf{Z}_{tmp} := \mathsf{Z}_R \setminus \mathsf{Z}_{Van} \setminus \mathsf{Z}_{Tan}$;
(16) $\quad \mathsf{Z}_{Tan} := \mathsf{Z}_{Tan} + \mathsf{Z}_{tmp}$;
(17) $\quad$ FOR $k \in \mathcal{A}ct^m$ DO
(18) $\quad\quad \mathsf{Z}_{new} := p\mathtt{ZAbstract}(\widetilde{\mathsf{Z}_k} \cdot \mathsf{Z}_{tmp}, \mathcal{V}_\mathsf{s}, +) \setminus Z_R$;
(19) $\quad\quad \mathsf{Z}_U := \mathsf{Z}_U + \mathsf{Z}_{new}\{\mathcal{V}_\mathsf{s} \leftarrow \mathcal{V}_\mathsf{t}\} \setminus Z_R$;
(20) $\quad$ END
(21) $\quad \mathsf{Z}_R := \mathsf{Z}_R + \mathsf{Z}_U$;
(22) END UNTIL $\mathsf{Z}_U = \emptyset$
(23) RETURN Z_R;

(B) Quasi dfs scheme

Algorithm C.3: Symbolic reachability analysis for models with immediate activities

`InitNewRound`()
(0) $\quad \mathsf{Z}_R := \mathtt{ReachabilityAnalysis}()$
/ Test composed states for triggering new immediate model behaviour */*
(1) $\quad$ FOR $k \in \mathcal{A}ct^i$ DO
(2) $\quad\quad \mathsf{Z}_{tmp} := \mathsf{Z}_R \setminus \mathsf{E}_k$;
(3) $\quad\quad$ WHILE $\mathsf{Z}_{tmp} \neq \emptyset$ DO
(4) $\quad\quad\quad \mathsf{Z}_s \xleftarrow{\vec{s}} \mathsf{Z}_{tmp}$;
(5) $\quad\quad\quad \vec{s} := \mathcal{E}^{-1}(\mathtt{Encode}^{-1}(\mathsf{Z}_s))$;
(6) $\quad\quad\quad \mathsf{Z}_s := p\mathtt{ZAbstract}(\mathsf{Z}_s, \mathcal{V}_\mathsf{s}^{\mathsf{I}_k}, +)$;
(7) $\quad\quad\quad \mathsf{Z}_{tmp} := \mathsf{Z}_{tmp} \setminus \mathsf{Z}_s$;
(8) $\quad\quad\quad$ IF $\vec{s}\,[> k$ THEN $\mathbb{S}$-Buffer $\longleftarrow (\vec{s}, \{k\})$; $\mathsf{Z}_{Van} := \mathsf{Z}_{Van} + \mathsf{Z}_R \cdot \mathsf{Z}_s$;
(9) $\quad\quad$ END
(10) $\quad$ END
/ Test composed states for triggering new Markovian model behaviour */*
(11) $\quad \mathsf{Z}_{Tan} := \mathsf{Z}_R \setminus \mathsf{Z}_{Van}$
(12) $\quad$ FOR $k \in \mathcal{A}ct^m$ DO
(13) $\quad\quad \mathsf{Z}_{tmp} := \mathsf{Z}_R \setminus \mathsf{E}_k$;
(14) $\quad\quad$ WHILE $\mathsf{Z}_{tmp} \neq \emptyset$ DO
(15) $\quad\quad\quad \mathsf{Z}_s \xleftarrow{\vec{s}} \mathsf{Z}_{tmp}$;
(16) $\quad\quad\quad \vec{s} := \mathcal{E}^{-1}(\mathtt{Encode}^{-1}(\mathsf{Z}_s))$;
(17) $\quad\quad\quad \mathsf{Z}_s := p\mathtt{ZAbstract}(\mathsf{Z}_s, \mathcal{V}_\mathsf{s}^{\mathsf{I}_k}, +)$;
(18) $\quad\quad\quad \mathsf{Z}_{tmp} := \mathsf{Z}_{tmp} \setminus \mathsf{Z}_s$;
(19) $\quad\quad\quad$ IF $\vec{s}\,[> k$ THEN $\mathbb{S}$-Buffer $\longleftarrow (\vec{s}, \{k\})$; $\mathsf{Z}_{Tan} := \mathsf{Z}_{Tan} + \mathsf{Z}_R \cdot \mathsf{Z}_s$;
(20) $\quad\quad\quad \mathsf{E}_k := \mathsf{E}_k + \mathtt{Encode}(\mathcal{E}(\vec{s}_{D_l}), 1, \mathcal{V}_\mathsf{s}^{\mathsf{D}_l})$;
(21) $\quad\quad$ END
(22) $\quad$ END
(23) $\quad$ RETURN ;

Algorithm C.4: Re-initialization when immediate activities are present

References

[ADD97] *Formal Methods in System Design: Special Issue on Multi-terminal Binary Decision Diagrams*, Volume 10, No. 2-3, April - May 1997.

[Age74] T. Agerwala. A complete model for representing the coordination of asynchronous processes. 32, Baltimore: Johns Hopkins University, Hopkins Computer Science Program, Research, july 1974.

[Ake78] S.B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

[AKN+00] L. de Alfaro, M. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic Model Checking for Probabilistic Processes using MTBDDs and the Kronecker Representation. In *TACAS'2000, LNCS 1785*, pages 395–410, 2000.

[Bai05] Ch. Baier. Binäre Entscheidungsdiagramme, 2005. Skriptum zur Vorlesung im Sommersemester 2005.

[BCD+95] G. Balbo, G. Conte, S. Donatelli, G. Franceschinis, M. Ajmone Marsan, and M. Ajmone Marsan. *Modelling with Generalized Stochastic Petri Nets*. JOHN WILEY & SONS, 1995.

[BCL91] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic Model Checking with Partitioned Transition Relations. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, pages 49–58, Edinburgh, Scotland, 1991. North-Holland.

[BFG+93] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and Their Applications. In *IEEE /ACM International Conference on CAD*, pages 188–191, Santa Clara, California, 1993. IEEE Computer Society Press.

[BG05] P. Bazan and R. German. Approximate Analysis of Stochastic Models by Self-Correcting Aggregation. In QEST'93 [QES05], pages 134–144.

[BH01] A. Bell and B. Haverkort. Serial and Parallel Out-Of-Core Solution of Linear Systems arising from Generalised Stochastic Petri Nets. In *Proc. of High Performance Computing 2001*, Seattle USA, April 2001.

[BHH+04] Ch. Baier, B. Haverkort, H. Hermanns, J.-P. Katoen, and M. Siegle, editors. *Validation of Stochastic Systems*, LNCS 2925, Dagstuhl (Germany), 2004. Springer.

[Boo52] G. Boole. *The Mathematical Analysis of Logic*. Collected logical works / George Boole. Open Court Publ. Comp., La Salle, Ill. (USA), 1952. The original appeared 1847.

[Bry86] R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Bry92] R.E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, September 1992.

[Buc91] P. Buchholz. *Die strukturierte Analyse Markovscher Modelle*. PhD thesis, Universität Dortmund, Dortmund (Germany), 1991.

[Buc94] P. Buchholz. Exact and ordinary lumpability in finite markov chains. *Journal of Applied Probability*, 31(1):59–75, March 1994.

[Buc06] P. Buchholz. Structured Analysis Techniques for Large Markov Chains. In *1'st Workshop on Tools for solving structured Markov Chains*, New York, NY, USA, 2006. ACM Press.

[CB06] F. Ciesinski and Ch. Baier. LiQuor: A tool for Qualitative and Quantitative Linear Time analysis of Reactive Systems. In *Proceedings of the 3'rd International Conference on Quantitative Evaluation of Systems (QEST'06)*, pages 131–132, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[CBC+93] G. Ciardo, A. Blakemore, P. F. J. Chimento, J. K. Muppala, and K. S. Trivedi. Automated generation and analysis of Markov reward models using Stochastic Reward Nets. *IMA Volumes in Mathematics and its Applications*, 48:145–191, 1993.

[CFM+93] E. M. Clarke, M. Fujita, P. McGeer, J. Yang, and X. Zhao. An Efficient Data Structure for Matrix Representation. In *Proc. of the International Workshop on Logic Synthesis (IWLS)*, 1993.

[CLS00] G. Ciardo, G. Luettgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In M. Nielsen and D. Simpson, editors, *Proc. of the 21'st Int. Conf. on Application and Theory of Petri Nets (ICATPN'00)*, LNCS 1825, pages 103–122, Aarhus, Denmark, June 2000. Springer.

[CLS01] G. Ciardo, G. Luettgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In T. Margaria and W. Yi, editors, *Proc. of the 7'th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01), Genova, (Italy)*, LNCS 2031, 2001.

[CM99a] G. Ciardo and A. S. Miner. A data structure for the efficient Kronecker solution of GSPNs. In P. Buchholz, editor, *Proc. 8th Int. Workshop on Petri Nets and Performance Models (PNPM'99)*, pages 22–31. IEEE Comp. Sc. Press., September 1999.

[CM99b] G. Ciardo and A. S. Miner. Efficient reachability set generation and storage using decision diagrams. In H. Kleijn and S. Donatelli, editors, *Proc. of the 20'th Int. Conf. on Application and Theory of Petri Nets (ICATPN'99), Williamsburg (VA, USA)*, LNCS 1639, pages 6–25. Springer, June 1999.

[CMF+93] E. M. Clarke, K.L. McMillian, M. Fujita, J. Yang, and X. Zhao. Spectral Transforms of Large Boolean Functions with Applications to Technology Mapping. In DAC'93 [DAC93], pages 54–60.

[CMS03] G. Ciardo, R. Marmorstein, and R. Siminiceanu. Saturation unbound. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 2619, 2003.

[CT93] G. Ciardo and K. Trivedi. A decomposition approach for stochastic reward net models. *Performance Evaluation*, 18(1):37–59, 1993.

[CT96] G. Ciardo and M. Tilgner. On the use of Kronecker operators for the solution of generalized stochastic Petri nets. Technical Report 96-35, Institute for Computer Applications in Science and Engineering, 1996.

[CY05] G. Ciardo and A. Jinqing Yu. Saturation-Based Symbolic Reachability Analysis Using Conjunctive and Disjunctive Partitioning. In *Charme 2005*, LNCS 3725, pages 146–161, 2005.

[DAC93] *Proc. of the 30th Design Automation Conference (DAC)*, Dallas (Texas), USA, June 1993. ACM / IEEE.

[DB98] R. Drechsler and B. Becker. *Graphen-basierte Funktionendarstellung*. Leitfäden der Informatik. B.G. Teubner, Stuttgart, 1998.

[DCC+02] D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. Doyle, W.H. Sanders, and P. Webster. The Moebius Framework and Its Implementation. *IEEE Transactions on Software Engineering*, 28(10):956–969, 2002.

[DCKS02] S. Derisavi, T. Courtney, P. Kemper, and W. H. Sanders. The Moebius State-level Abstract Functional Interface. In *Proc. of Performance Tools 2002: 12th Int. Conf. on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation*, pages 31–50, 2002.

[DKK02] I. Davies, W.J. Knottenbelt, and P.S. Kritzinger. Symbolic Methods for the State Space Exploration of GSPN Models. In *Proc. of the 12th Int. Conf. on Modelling Techniques and Tools (TOOLS 2002)*, pages 188–199. LNCS 2324, 2002.

[DKS03] S. Derisavi, P. Kemper, and W. H. Sanders. Symbolic State-space Exploration and Numerical Analysis of State-sharing Composed Models. In *Proc. Fourth Int. Conf. on Numerical Solution of Markov Chains*, pages 167–189, 2003.

[DKS05] S. Derisavi, P. Kemper, and W. H. Sanders. Lumping Matrix Diagram Representations of Markov Models. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2005)*, pages 742–751. IEEE Computer Society, 2005.

[EFT93] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for symbolic model checking in CCS. *Distributed Computing*, 6(3):155–164, 1993.

[FG88] Bennett L. Fox and Peter W. Glynn. Computing poisson probabilities. *Commun. ACM*, 31(4):440–445, 1988.

[FM97] M. Fujita and P. McGeer, editors. *Formal Methods in System Design: Special Issue on Multi-terminal Binary Decision Diagrams*, 1997. Vol. 10, No. 2/3.

[Fra99] E. Frank. Codierung und numerische Analyse von Transitionssystemen unter Verwendung von MTBDDs, Oktober 1999. Studienarbeit am IMMD VII, Universität Erlangen–Nürnberg, (Student Thesis).

[Ger00] R. German. *Performance Analysis of Communication Systems, Modeling with Non-Markovian Stochastic Petri Nets*. John Wiley & Sons, 2000.

[GLW00] G. Graf, M. Leberecht, and M. Walter. High Availability Commodity Computing - A CompactPCI-System Evaluation. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 4. CSREA Press, 2000.

[God95] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems. An Approach to the State-Explosion Problem*. PhD thesis, Université de Liege, 1995.

[Göt94] N. Götz. *Stochastische Prozessalgebren – Integration von funktionalem Entwurf und Leistungsbewertung verteilter Systeme*. PhD thesis, Friedrich-Alexander-Universität Erlangen–Nürnberg, Erlangen (Germany), 1994.

[Hac76] M. Hack. Decidability Questions for Petri Nets. Technical report, Cambridge: Massachusetts Institute of Technology, 1976.

[Har06] S. Harwarth. Computation of transient state probabilities and implementing Möbius' "state-level abstract functional interface" for the data structure ZDD, 2006. Master Thesis. University of the Federal Armed Forces Munich (Germany).

[HBB99] B.R. Haverkort, A. Bell, and H. Bohnenkamp. On the Efficient Sequential and Distributed Generation of very Large Markov Chains from Stochastic Petri Nets. In *Proc. of IEEE Petri Nets and Performance Models*, pages 12–21, 1999.

[Her98] H. Hermanns. *Interactive Markov Chains*. PhD thesis, Friedrich-Alexander-Universität Erlangen—Nürnberg, Erlangen (Germany), 1998.

[HHK+98] H. Hermanns, U. Herzog, U. Klehmet, V. Mertsiotakis, and M. Siegle. Compositional performance modelling with the TIPPtool. In R. Puigjaner, N. Savino, and B. Serra, editors, *10'th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'98)*, LNCS 1469, pages 51–62. Springer Verkag, September 1998.

[HHM98] H. Hermanns, U. Herzog, and V. Mertsiotakis. Stochastic Process Algebras - Between LOTOS and Markov Chains. *Computer Networks and ISDN Systems, 30 (9-10)*, pages 901–924, 1998.

[HHMR97] Holger Hermanns, Ulrich Herzog, Vassilis Mertsiotakis, and Michael Rettelbach. Exploiting stochastic process algebra achievements for generalized stochastic petri nets. In *PNPM '97: Proceedings of the 6th International Workshop on Petri Nets and Performance Models*, page 183, Washington, DC, USA, 1997. IEEE Computer Society.

[Hil94a] J. Hillston. *A Compositional Approach to Performance Modelling*. PhD thesis, University of Edinburgh, Edinburgh (UK), 1994.

[Hil94b] J. Hillston. The nature of synchronisation. In U. Herzog and M. Rettelbach, editors, *Proc. of the Second Int. Workshop on Process Algebras and Performance Modelling*, pages 51–70, Erlangen (Germany), 1994.

[HKN+03] H. Hermanns, M. Kwiatkowska, G. Norman, D. Parker, and M. Siegle. On the use of MTBDDs for performability analysis and verification of stochastic systems. *Journal of Logic and Algebraic Programming*, 56(1-2):23–67, 2003.

[HL94] G. Horton and S.T. Leutenegger. A multi-level solution algorithm for steady-state markov chains. In *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 191–200, New York, NY, USA, 1994. ACM Press.

[HMKS99] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi Terminal Binary Decision Diagrams to Represent and Analyse Continuous Time Markov Chains. In *Proc. of 3rd Int. Workshop on Numerical Solution of Markov Chains*, pages 188–207. Prensas Universitarias de Zaragoza, 1999.

[HW06] M. Hammer and M. Weber. To Store or not to Store" Reloaded: Reclaiming Memory on Demand. In *Proceedings of Formal Methods on Industrial Critical Systems 2006*, LNCS 4346, 2006.

[Inta] Computing cluster at the Technical University Munich. http://www.lrr.in.tum.de/Par/arch/infiniband/ClusterHW/cluster.html.

[Intb] LinuxRamLimits. http://www.spack.org/wiki/LinuxRamLimits.

[IT90] O. Ibe and K. Trivedi. Stochastic Petri net models of polling systems. *IEEE Journal on Selected Areas in Communications*, 8(9):1649–1657, 1990.

[JHK03] David N. Jansen, Holger Hermanns, and Joost-Pieter Katoen. A QoS-Oriented Extension of UML Statecharts. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20-24, 2003, Proceedings*, volume 2863 of *Lecture Notes in Computer Science*, pages 76–91. Springer, 2003.

[KL04] M. Kuntz and K. Lampka. Probabilistic methods in state space analysis. In Baier et al. [BHH$^+$04], pages 339–383.

[Kno99] W.J. Knottenbelt. *Parallel Performance Analysis of Large Markov Models*. PhD thesis, University of London, Imperial College, Dept. of Computing, 1999.

[KS02] M. Kuntz and M. Siegle. Deriving Symbolic Representations from Stochastic Process Algebras. In *Process Algebra and Probabilistic Methods (PAPM-PROBMIV'02)*, LNCS 2399, pages 1–22, 2002.

[KSW04] M. Kuntz, M. Siegle, and E. Werner. Symbolic Performance and Dependability Evaluation with the Tool CASPA. In *Proc. of EPEW*, pages 293–307. Springer, LNCS 3236, 2004.

[KVBSV98] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1-2):9–62, 1998.

[Lam00] K. Lampka. A stochastic process algebra interface for DNAmaca, April 2000. Student Thesis. IMMD VII, Friedrich-Alexander Universität Erlangen–Nürnberg.

[Lee59] C.Y. Lee. Representation of Switching Circuits by Binary-Decision Programs. *Bell Systems Technical Journal*, 38:985–999, July 1959.

[Lin98] C. Lindemann. *Performance Modelling with Deterministic and Stochastic Petri Nets*. Wiley and Sons, 1998.

[LS02] K. Lampka and M. Siegle. Symbolic Composition within the Moebius Framework. In *Proc. of the 2nd MMB Workshop*, pages 63–74, September 2002. Forschungsbericht der Universität Hamburg Fachbereich Informatik.

[LS03a] K. Lampka and M. Siegle. MTBDD-based activity-local state graph generation. In *Sixth Int. Workshop on Performability Modeling of Computer and Communication Systems (PMCCS6)*, pages 15–18, September 2003.

[LS03b] K. Lampka and M. Siegle. Symbolic Activity-Local State Graph Generation in the Context of Moebius. In *Proc. of the Satelite Workshop on Stochastic Petri Nets and related Formalisms at the 30'th Int. Colloquium on Automata, Languages and Programming, Eindhoven, Netherlands*, June 2003.

[LS06a] K. Lampka and M. Siegle. Activity-Local State Graph Generation for High-Level Stochastic Models. In *Meassuring, Modelling, and Evaluation of Systems 2006*, pages 245–264, April 2006.

[LS06b] K. Lampka and M. Siegle. Analysis of Markov Reward Models using Zero-supressed Multi-terminal decision diagramms. In *Proceedings of VALUETOOLS 2006 (CD-edition)*, October 2006.

[LSW06] K. Lampka, M. Siegle, and M. Walter. An easy-to-use, efficient tool-chain to analyze the availability of telecommunication equipment. In *Proceedings of Formal Methods on Industrial Critical Systems 2006*, LNCS 4346, pages 35–50, 2006.

[Meh04] R. Mehmood. *Disk-based techniques for efficient solution of large Markov chains*. PhD thesis, University of Birmingham, University of Birmingham (U.K.), October 2004.

[Min93] S. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In DAC'93 [DAC93], pages 272–277.

[Min96] S. Minato. *Graph-based Representation of discrete Functions*, pages 1–28. Volume 1 of Sasao and Fujita [SF96], 1996.

[Min01] A. Miner. Efficient solution of GSPNs using Canonical Matrix Diagrams. In R. German and B. Haverkort, editors, *Proc. of the 9'th Int. Workshop on Petri Nets and Performance Models (PNPM'01)*, pages 101–110, Aachen, Germany, September 2001.

[Min04] A. Miner. Saturation for a general class of models. In *Proc. of the First Int. Conf. on the Quantitative Evaluation of Systems (QEST'04)*, pages 282–291. IEEE Computer Society Press, 2004.

[MP04] A. Miner and D. Parker. Symbolic Representations and Analysis of Large State Spaces. In Baier et al. [BHH+04], pages 296–338.

[MS91] J. F. Meyer and W. H. Sanders. A unified Approach for specifying Measures of Performance, Dependability, and Performability. In *Dependable Computing for Critical Applications*, Vol. 4, pages 215–237. Springer-Verlag, 1991.

[MS92] L. M. Malhis and W. H. Sanders. Dependability evaluation using composed SAN-based reward models. *Journal of Parallel and Distributed Computing*, 15:238–254, 1992.

[MT98] Ch. Meinel and Th. Theobald. *Algorithms and Data Structures in VLSI-Design*. Springer, Berlin, New York, et. al., 1998.

[Oba98] W.D. Obal II. *Measure-adaptive State-Space Construction Methods*. PhD thesis, University of Arizona, Arizona (USA), 1998.

[Oss06] J. Ossowski. The JINC BDD package. September 2006.

[Par02] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, Birmingham (U.K.), 2002.

[PC98] E. Pastor and J. Cortadella. Efficient Encoding Schemes for Symbolic Analysis of Petri Nets. In *In Proc. of Design, Automation and Test in Europe 1998*, pages 790–795, February 1998.

[Pla85] Brigitte Plateau. On the stochastic structure of parallelism and synchronization models for distributed algorithms. In *SIGMETRICS '85: Proceedings of the 1985 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 147–154, New York, NY, USA, 1985. ACM Press.

[PRC97] E. Pastor, O. Roig, and J. Cortadella. Symbolic Petri Net Analysis using Boolean Manipulation, 1997. Technical Report of Departament Arquitectura de Computadors (UPC) DAC/UPC Report No. 97/8.

[PRCB94] E. Pastor, O. Roig, J. Cortadella, and R.M. Badia. Petri Net Analysis Using Boolean Manipulation. In R. Valette, editor, *Proc. of the 15'th Int. Conf. on Application and Theory of Petri Nets (APN'94), Zaragosa, Spain*, LNCS 815, pages 416–435. Springer, June 1994.

[Pri] PRISM web page. http://www.cs.bham.ac.uk/∼dxp/prism/.

[QES05] *Second International Conference on the Quantitative Evaluation of Systems (QEST'05)*, Torino (Italy), September 2005. IEEE Computer Society.

[San88] W.H. Sanders. *Construction and solution of performability models based on stochastic activity networks*. PhD thesis, University of Michigan, 1988.

[Sas96] T. Sasao. *Ternary Decision Diagrams and their Applications*, pages 268–292. Volume 1 of Sasao and Fujita [SF96], 1996.

[SF96] T. Sasao and M. Fujita, editors. *Representations of Discrete Functions*, volume 1. Kluwer Academic Publishers, Dordrecht The Netherlands, 1996.

[Sha00] C.S. Shannon. *Eine symbolische Analyse von Relaisschaltkreisen*. Verlag Brinkmann + Bose, 2000. The article originally appeared with the title: *A Symbolic Analysis of Switching Circuits* in Transactions AIEE, 57 (1938), 713.

[Sie95] M. Siegle. *Beschreibung und Analyse von Markovmodellen mit großem Zustandsraum*. PhD thesis, Friedrich-Alexander-Universität Erlangen–Nürnberg, Erlangen (Germany), 1995.

[Sie98] M. Siegle. Compact representation of large performability models based on extended BDDs. In *Fourth Int. Workshop on Performability Modeling of Computer and Communication Systems (PMCCS4)*, pages 77–80, Williamsburg, VA, September 1998.

[Sie01] M. Siegle. Advances in model representation. In Luca de Alfaro and Stephen Gilmore, editors, *Proc. of the Joint Int. Workshop, PAPM-PROBMIV 2001, Aachen (Germany)*, LNCS 2165, pages 1–22. Springer, September 2001.

[Sie02] M. Siegle. *Behaviour analysis of communication systems: Compositional modelling, compact representation and analysis of performability properties*. Shaker Verlag, Aachen, 2002.

[Sma] SMART web page. http://www.cs.ucr.edu/∼ciardo/SMART.

[Som98] F. Somenzi. CUDD: Colorado University Decision Diagram Package, Release 2.3.0. User's Manual and Programmer's Manual, September 1998.

[SS03] R. Sangireddy and A.K. Somani. High-Speed IP Routing With Binary Decision Diagrams based HW Address Lookup-Engine. *IEEE Journal on Selected Areas in Communications*, 21(4):513–521, 2003.

[Ste94] W. J. Stewart. *An Introduction to the solution of Markov Chains*. Princeton University Press, Princeton, NJ (USA), 1994.

[Web02] M. Weber. Eine Komponente zur effizienten Steuerung des reduzierten Erreichbarkeitsgraphen von stochastischen Petrinetzen, 2002. Diplomarbeit angefertigt am IMMD VII, Friedrich-Alexander Universität Erlangen-Nürnberg, (Master Thesis).

[Wei05] W. Weiss. Parallele Erzeugung symbolischer Repräsentation von Zustandsgraphen auf der Basis von "Zero-suppressed" Multi-terminalen Binären Entscheidungsdiagrammen, March 2005. Diplomarbeit angefertigt am IMMD VII, Friedrich-Alexander Universität Erlangen–Nürnberg (Master Thesis).

[WL91] M. Woodside and Y. Li. Performance Petri net analysis of communications protocol software by delay-equivalent aggregation. In *In Proc. of the 4th Int. Workshop on Petri Nets and Performance Models (PNPM)*, pages 64–73, 1991.

[WT05] M. Walter and C. Trinitis. OpenSESAME: Simple but Extensive Structured Availability Modeling Environment. In QEST'93 [QES05], pages 253– 254.

[Zim05] D. Zimmermann. Implementierung von Verfahren zur Lösung dünn besetzter linearer Gleichungssysteme auf Basis von Zero-suppressed Multi-terminalen Binären Entscheidungsdiagramme, 2005. Diplomarbeit angefertigt an der Universität der Bundeswehr Neubiberg (Germany) (Master Thesis).

German Translations

Ein symbolischer Ansatz für die Zustandsgraph-basierte Analyse von hochsprachlichen Markov Reward Modellen

Deutsche Übersetzung ausgewählter Teile,
gemäss Promotionsordnung der Technischen Fakultät
der Universität Erlangen-Nürnberg

vorgelegt von
Kai Matthias Lampka

Erlangen, März 2007

Inhaltsverzeichnis des deutschsprachigen Teils

1 Übersetzung der Kurzfassung 1

2 Übersetzung des Inhaltsverzeichnisses 3

3 Übersetzung des Kapitels "Introduction" (Einführung) 7
3.1 Motivation 7
3.2 Zustandsraumexplosionsproblem und verwandte Ansätze 8
3.3 Stand der symbolischen Techniken 11
3.4 Eigener Beitrag 13
3.5 Aufbau der Arbeit 14

4 Übersetzung des Kapitels "Conclusion" (Schlussbetrachtung) 15
4.1 Zusammenfassung 15
4.1.1 Zero-suppressed multi-terminale BDDs 15
4.1.2 Aktivitäts/Reward-lokales Schema 16
4.1.3 Berechnung der Zustandswahrscheinlichkeiten 17
4.1.4 Berechnung der Performabilitätsmaße 18
4.2 Vorteile von *p*ZDDs und dem Aktivitäts/Reward-lokalen Schema 18
4.3 Zukünftige Arbeiten 19

1

Übersetzung der Kurzfassung

Markov Reward Modelle, wie sie in dieser Arbeit behandelt werden, sind kompakt durch Markovsche Erweiterungen bekannter hochsprachlicher Modellbeschreibungsformalismen beschrieben. Um mit numerischen Verfahren die "Performabilitätmaße", das sind Leistungs- und Verfügbarkeitsmaße, von hochsprachlich spezifizierten Modellen zu berechnen, müssen letztere in eine Repräsentationsform niederer Stufe gebracht werden, wobei die im hochsprachlichen Modell enthaltene Nebenläufigkeit explizit ausgedrückt wird. Diese Transformation, die ein hochsprachliches Modell auf ein Zustands/Transitionssystem abbildet, allg. auch als Zustandsgraph (ZG) bezeichnet, kann deswegen zu einem exponentiellen Wachstum in der Anzahl der Systemzustände führen. Dieses Problem ist als das sog. Zustandsraumexplosionsproblem wohl bekannt. Entscheidungsdiagramme (DD) haben sich als sehr nützlich erwiesen, wenn es um die Repräsentation extrem großer ZG geht. Mit ihrer Hilfe lassen sich nun größere und komplexere Modelle und somit auch Systeme analysieren. Um die zeitgenössischen symbolischen Techniken jedoch anwenden zu können, müssen die zu analysierenden Modelle entweder eine bestimmte kompositionelle Struktur besitzen und/oder ein bestimmter Modellbeschreibungsformalismus verwendet worden sein. Diese Einschränkungen werden in dieser Arbeit beseitigt, wobei jedoch die Anzahl der Zustände, deren Wahrscheinlichkeit es zu berechnen gilt, der limitierende Faktor bei der Analyse bleibt.

Zur symbolischen Darstellung (stochastischer) ZG erweitert diese Arbeit "zero-suppressed" binäre Entscheidungsdiagramme zu "zero-suppressed" multi-terminalen binären Entscheidungsdiagrammen (ZDD). Um die korrekte Pseudo-Boolesche Funktion, die durch den Graphen eines ZDDs dargestellt werden soll, abzuleiten, muß die Menge der Funktionsvariablen des ZDDs bekannt sein. Als Konsequenz ergibt sich, dass i. Allg. die innerhalb einer gemeinsamen Umgebung, wie sie durch bekannte DD-Programmpakete bereitgestellt werden, allokierten ZDD-Knoten ihre Eindeutigkeit verlieren. Um dieses Problem zu lösen entwickelt diese Arbeit das Konzept der *partiell gemeinsamen* ZDDs (*p*ZDDs), welches ZDD-Knoten um Funktionsvariablen erweitert. Es wird gezeigt, dass Entscheidungsdiagramme diesen Typs eine kanonische Form der Darstellung von Pseudo-Booleschen Funktionen sind. Um effizient mit *p*ZDDs arbeiten zu können, wird ein breites Spektrum an (symbolischen) Algorithmen entwickelt. Diese Algorithmen sind so konzipiert, dass sie eine Implementierung von *p*ZDDs in gewöhnlichen, gemeinsamen DD-Umgebungen erlauben.
Besitzt ein Modellbeschreibungsformalismus keine symbolische Semantik, so können symbolische Repräsentationen der annotierten Zustands/Transitionssysteme seiner hochsprachlichen Modellbeschreibungen nur dadurch gewonnen werden, dass man die hochsprachlichen Systemmodelle explizit ausführt. Um dies speicher- und zeiteffizient zu tun, verwendet diese Arbeit nur lokale Informationen bzgl. der hochsprachlichen Modellkonstrukte. Der so neu entstandene semi-symbolische Ansatz, den wir als *Aktivitäts/Reward-lokalen* Ansatz bezeichnen, umfasst nun die folgenden vier Schritte: (a) Das *Aktivitäts-lokale* Schema zur Generierung der symbolischen Repräsentation des ZG eines hochsprachlichen Modells. Da dieser Ansatz nicht alle Systemzustände explizit erzeugt, ist die Verwendung eines symbolischen Kompositionsschemas nötig. Das hier entwickelte Kompositionsschema erzeugt dabei den potentiellen ZG, seine Einschränkung auf erreichbare Elemente kann allerdings effizient mittels einer symbolischen Erreichbarkeitsanalyse herbeigeführt werden, wobei diese Arbeit einen neuen, "quasi" tiefe-suchenden Algorithmus einführt. (b) Das *Reward*-lokale Schema um symbolische Repräsentationen der Rewardfunktionen des jeweiligen hochsprachlichen Modells zu erzeugen. Analog zur obigen Vorgehensweise werden dabei die Rewardfunktionen explizit ausgeführt, um die Rewardwerte der Zustände und Transitionen zu bestimmen. Um die Anzahl der expliziten Zustandsbesuche gering zu halten, werden jedoch erneut nur

lokale Informationen verwendet. (c) Zur Berechnung der Zustandswahrscheinlichkeiten führt diese Arbeit eine ZDD-basierte Variante des hybriden Lösungsverfahrens, wie es im Kontext anderer symbolischer Datentypen entwickelt wurde, ein. (d) Auf Basis der symbolisch dargestellten Rewardfunktionen und den Zustandswahrscheinlichkeiten, ist man nun in der Lage, die durch den Benutzer spezifizierten Leistungsmaße des hochsprachlichen Modells zu bestimmen, wobei dies mittels eines neuen symbolischen Algorithmus' realisiert wird.

Da der *Aktivitäts/Reward-lokale* Ansatz auf einer expliziten Ausführung des hochsprachlichen Modells beruht, die in der Praxis wahrscheinlich nur partiell durchgeführt werden muss, ist er nicht auf einen bestimmten Modellbeschreibungsformalismus beschränkt. Basierend auf einem neuen symbolischen Kompositionsschema und im Gegensatz zu anderen symbolischen Ansätzen, ist er auch dann einsetzbar, wenn die hochsprachlichen Modelle weder kompositionell konstruiert, noch eine dekomponierbare Struktur bestimmter Art besitzen. Diese Arbeit entwickelt somit nicht nur einen neuen Typ von Entscheidungsdiagramm und Algorithmen zu seiner effizienten Manipulation, sondern auch einen universellen Ansatz zur ZG-basierten Analyse von hochsprachlichen Markov Reward Modellen mit sehr großen ZG.

2

Übersetzung des Inhaltsverzeichnisses

Verzeichnis der Abbildungen ... IV

Verzeichnis der Algorithmen ... V

Verzeichnis der Tabellen ... VI

1 Einleitung ... 1
- 1.1 Motivation ... 1
- 1.2 Zustandsraumexplosionsproblem und verwandte Ansätze ... 2
- 1.3 Stand der symbolischen Techniken ... 4
- 1.4 Eigener Beitrag ... 7
- 1.5 Aufbau der Arbeit ... 7

2 Hintergrund ... 9
- 2.1 Aufbau des Kapitels ... 9
- 2.2 Markov Theorie ... 9
  - 2.2.1 Markov Reward Modelle mit kontinuierlicher Zeit (MRM) ... 9
  - 2.2.2 Numerische Lösung von MRM ... 11
  - 2.2.3 Reduktionstechniken ... 16
  - 2.2.4 Zustands/Transitionssysteme ... 19
- 2.3 Hochsprachliche Markov Reward Modelle ... 20
  - 2.3.1 Hochsprachliche Modellbeschreibungstechniken ... 21
  - 2.3.2 Spezifikation der Performabilitätsmaße ... 22
  - 2.3.3 Komposition hochsprachlicher Modellbeschreibungen ... 23
  - 2.3.4 Abbildung hochsprachlicher Modelle auf MRM ... 24
- 2.4 Nicht-kompositionelle Konstruktion des Zustandsgraphen (ZG) ... 24
- 2.5 Kompositionelle Konstruktion des Zustandsgraphen ... 25
  - 2.5.1 Grundlagen ... 26
  - 2.5.2 Komposition des ZG im Falle der reinen Verschränkung ... 26
  - 2.5.3 Komposition des ZG im Falle der Aktivitätssynchronisation ... 27
  - 2.5.4 Komposition des ZG im Falle von gemeinsamen Zustandsvariablen ... 28
  - 2.5.4 Komposition des ZG im Falle von replizierten Submodellen ... 29
  - 2.5.5 Beschränkung Kronecker-Operator basierter Kompositionsschemata ... 29

3 Zero-suppressed multi-terminale BDDs: Konzepte, Algorithmen und Anwendung ... 31
- 3.1 Aufbau des Kapitels ... 31
- 3.2 Binäre Entscheidungsdiagramme und Erweiterungen ... 32
  - 3.2.1 Binäre Entscheidungsdiagramme (BDDs) ... 32
  - 3.2.2 Zero-suppressed BDDs (z-BDDs) ... 37
  - 3.2.3 Multi-terminale BDDs (ADDs) ... 38
  - 3.2.4 Zero-suppressed multi-terminale BDDs (ZDDs) ... 39
- 3.3 Partiell gemeinsame ZDDs (pZDDs) ... 40
  - 3.3.1 Definitionen ... 41
  - 3.3.2 Kanonizität von pZDDs ... 43
- 3.4 Operationen auf pZDDs ... 45
  - 3.4.1 Vorbemerkung ... 45

3.4.2 Anwendung binärer Operatoren auf *p*ZDDs 46
3.4.3 Varianten des *p*`ZApply`-Algorithmus 52
3.4.4 Umbenennung von Variablen 54
3.4.5 Der *p*`ZRestrict`-Operator 54
3.4.6 Der *p*`ZAbstract`-Operator 54
3.5 Anwendungen 56
3.5.1 ZDD-basierte Darstellung von Mengen und Relationen 57
3.5.2 ZDD-basierte Darstellung von Matrizen 59
3.5.3 Erweiterung von ZDDs zur effizienten Berechnung von Matrix-Vektor-Produkten 66
3.5.4 Über DD-basierte Matrixdarstellungen hinausgehendes Konzept 69
3.6 Vergleichbare Arbeiten und eigener Beitrag 69

4 Das Aktivitäts/Reward-lokale Schema: Symbolische, Zustandsgraph-basierte Analyse von hochsprachlichen Markov Reward Modellen 73
4.1 Aufbau des Kapitels 73
4.2 Modellwelt 74
4.2.1 Statische Eigenschaften 74
4.2.2 Dynamische Eigenschaften 75
4.2.3 Abgeleitete Eigenschaften 81
4.2.4 Beschränktheit der Modelle 86
4.3 Das Aktivitäts-lokale Schema: Generierung symbolischer Repräsentationen von Zustandsgraphen 86
4.3.1 Hauptroutine 86
4.3.2 Explizite Zustandsgraphgenerierung und -kodierung 88
4.3.3 Symbolische Zustandsgraphkomposition 89
4.3.4 Symbolische Erreichbarkeitsanalyse 89
4.3.5 Re-Initialisierungsschema 91
4.3.6 Beispiel 91
4.4 Vollständigkeit und Korrektheit des Schemas 94
4.4.1 Generierungsschema 94
4.4.2 Kompositionsschema 96
4.4.3 Erreichbarkeitsanalyse 98
4.5 Berechnung der Performabilitätsmaße 99
4.5.1 Berechnung der Zustandswahrscheinlichkeiten 100
4.5.2 Das Reward-lokale Schema: Generierung symbolischer Repräsentationen von Rewardfunktionen . . 100
4.5.3 Berechnung der Momente von Performanzvariablen 102
4.6 Erweiterung des grundlegenden Aktivitäts-lokalen Schemas 103
4.6.1 Behandlung von explizit modellierten Symmetrien 103
4.6.2 Behandlung von instantanen Aktivitäten 106
4.7 Verwandte Arbeiten und eigener Beitrag 108
4.7.1 Voll-symbolische Techniken 111
4.7.2 Semi-symbolische Techniken 112
4.7.3 Semi-symbolische, kompositionelle und submodel-interdependente Verfahren 114
4.7.4 Symbolische Algorithmen zur Generierung der Menge der erreichbaren Zustände 116
4.8 Vorab veröffentlichtes Material 117

5 Empirische Bewertung . . . 119
5.1 Aufbau des Kapitels . . . 119
5.2 Vorbemerkung . . . 120
5.2.1 Verwendete "Benchmark"-Modelle . . . 121
5.2.2 Darstellung der gesammelten Laufzeitdaten . . . 124
5.2.3 Plattform . . . 124
5.2.4 Vergleiche . . . 124
5.3 Bewertung des Aktivitäts-lokalen Zustandsgraph-Generierungsschemas . . . 126
5.3.1 Vergleich von ADD- und ZDD-basierten Zustandsgraph-Generatoren . 126
5.3.2 Bewertung des neuen symbolischen Algorithmus zur Erreichbarkeitsanalyse . . . 128
5.3.3 Signifikanz der Variablenordnung . . . 130
5.4 Vergleich der symbolischen Zustandsgraph-Generierungstechniken . . . 131
5.4.1 Vergleich mit voll-symbolischen Methoden . . . 131
5.4.2 Vergleich mit anderen semi-symbolischen Methoden . . . 137
5.5 Bewertung der ZDD-basierten Löser . . . 142
5.5.1 Vergleich ADD- und ZDD-basierter numerischer Löser . . . 142
5.5.2 Wahl der Block- und "Sparse"-Ebene . . . 144
5.5.3 Signifikanz der Variablenordnung . . . 146
5.6 Vergleich mit anderen Lösern . . . 147
5.6.1 Vergleich mit den "Sparse-"Matrix-basierten Lösern von Möbius . . . 147
5.6.2 Vergleich mit den Lösern von Smart . . . 149
5.7 Fallstudie: Telekommunikations-Diensterbringungssystem . . . 151
5.7.1 Systembeschreibung . . . 151
5.7.2 Modellbewertung . . . 152
5.8 Vorab veröffentlichtes Material . . . 154

6 Schlußbetrachtung . . . 155
6.1 Zusammenfassung . . . 155
6.1.1 Zero-suppressed multi-terminale BDDs . . . 155
6.1.2 Aktivitäts/Reward-lokales Schema . . . 156
6.1.3 Berechnung der Zustandswahrscheinlichkeiten . . . 157
6.1.4 Berechnung der Performabilitätsmaße . . . 157
6.2 Vorteile von *p*ZDDs und dem Aktivitäts/Reward-lokalen Schema . . . 158
6.3 Zukünftige Arbeiten . . . 159

A Anhang: Mathematischer Hintergrund . . . 161
A.1 Boolesche Funktionen . . . 161
A.2 Pseudo-Boolesche Funktionen . . . 162
A.3 Kronecker-Operatoren . . . 162
A.4 Notation des Modus ponens . . . 163
A.5 Pseudo-Code und verwandte Notation . . . 163

B Anhang: Algorithmen für BDDs und abgeleitete Typen . . . 164

C Anhang: Algorithmen zur Handhabung von Modellen mit instantanen Aktivitäten . . . 165

Literaturverzeichnis . . . 167

Deutsche Übersetzungen . . . 173

3

Übersetzung des Kapitels “Introduction” (Einführung)

3.1 Motivation

Es ist allgemein bekannt, dass komplexe Hardware- und Softwaresysteme Teil des alltäglichen Lebens geworden sind. Da man zunehmend von diesen Systemen abhängig wird, ist es wichtig sicherzustellen, dass diese korrekt arbeiten und hohen Ansprüchen bzgl. Funktionalität, Leistung und Verfügbarkeit genügen. Oftmals ist es jedoch schwer oder gar nicht möglich, die Daten zu erheben, die man zur Bestimmung der Leistungsfähigkeit und Verfügbarkeit (= Performabilität) eines Systems benötigt, so dass Systemmessungen und das Systemtesten als Evaluationsmethode nicht in Betracht gezogen werden können. In solchen Fällen ist man dann darauf beschränkt, ein (mathematisches) Systemmodell zu analysieren und nicht das eigentliche System selbst. Der Vorteil einer solchen (formalen) Vorgehensweise ist leicht ersichtlich: Die modellbasierte Analyse erlaubt es, die Funktionalität und das quantitative Verhalten eines nicht notwendigerweise existierenden Systems zu bestimmen; so dass Korrektheit und Performabilität eines Systementwurfs bereits in einer frühen (Weiter-) Entwicklungsphase überprüft und somit kostenintensive Fehlentwicklungen vermieden werden können.

Annotierte Zustands/Transitionssysteme (*ST*-Systeme) bilden einen adäquaten Rahmen um komplexes Systemverhalten formal zu beschreiben. Jedoch sind heutige Hard- und Softwaresysteme oftmals parallel und sogar verteilt, so dass eine detaillierte Systembeschreibung als *ST*-System durch die Größe eines solchen behindert, wenn nicht schlichtweg unmöglich wird. Formale Beschreibungsmethoden, wie sie in den letzten Dekaden entwickelt wurden, haben gezeigt, dass sie mächtige Werkzeuge zum Erstellen kompakter Systembeschreibungen sind. Lässt man nun noch ein stochastisches Konzept der Zeit und die Beschreibung von Kosten und/oder Erträgen in die Modellbeschreibung einfließen, so erhält man das, was man gemeinhin als stochastisches hochsprachliches Performabilitätsmodell bezeichnet. In Abhängigkeit des erstellten hochsprachlichen Modells, der verwendeten hochsprachlichen beschreibungdmethode, und der Wahrscheinlichkeitsverteilung zur Beschreibung der zeitl. Verzögerung zwischen den Folgezuständen, kann man die gewünschten Maße entweder analytisch/numerisch oder empirisch bestimmen. Die Bestimmung von Maßen mittels einer analytischen Herangehensweise, d.h. durch Auswertung eines geschlossenen Ausdrucks, ist im wesentlichen auf bestimmte Klassen von Warteschlangennetzen beschränkt. Die empirische oder numerische Herangehensweise hingegen, erzwingt die partielle oder komplette Erzeugung des Zustandsgraphen des zu analysierenden Modells. Im Gegensatz zu der empirischen Methode, wie sie durch die System(modell)simulation verwirklicht wird und bei der nur Spuren des Systemverhaltens erzeugt werden, erlauben hochsprachliche Markov Reward Modelle eine exhaustive Analyse des Systemverhaltens. Der Vorteil der exhaustiven Analyse durch vollständige Erzeugung des zugrundeliegenden *ST*-Systems geht jedoch mit dem Nachteil einher, dass zeitlich verzögert Zustandsübergänge innerhalb des Modells nur nach Ablauf einer Zeitspanne auftreten können, deren Länge durch eine exponentiell verteilte Zufallsvariable beschrieben wird.

Markov Reward Modelle, wie sie in dieser Arbeit behandelt werden, sind kompakt durch Markovsche Erweiterungen von bekannten hochsprachlichen Modellbeschreibungsmethoden, wie bspw. durch stochastische Petri-Netze (SPN), stochastische Aktivitätsnetze (SAN) und stochastische Prozessalgebren (SPA), um nur einige zu nennen, beschrieben. Um ein beliebiges Modell dieser Art analysieren zu können, muss es zunächst in ein endliches, stochastisches *ST*-System transformiert werden, wobei dieses auch als niedere Darstellungsform oder als Zustandsgraphen bezeichnet wird. Das *ST*-System kann nun direkt als Markov Reward Mo-

dell interpretiert werden (im mathematischen Sinne). Die Theorie der Modelle dieser Art ist wohl bekannt, und sie erlaubt die Verteilung der Wahrscheinlichkeiten auf der Menge von Systemzuständen numerisch zu berechnen, wobei man die gewünschten Performabilitätsmaße durch Aggregation der Zustandswahrscheinlichkeiten erhält.

3.2 Zustandsraumexplosionsproblem und verwandte Ansätze

Der erste Schritt zur Analyse eines hochsprachlichen Markov Reward Modells ist die Generierung des Zustandsgraphen. Bei diesem Schritt ergibt sich das bekannte Zustandsraumexplosionsproblem.

Zustandsraumexplosion

Die Nebenläufigkeit von Aktivitäten muss im Fall der Modelltransformation explizit ausgedrückt werden. Die sog. “interleaving” Semantik, wie sie im Kontext von hochsprachlichen Standardmarkovmodellbeschreibungsmethoden Verwendung findet, ergibt eine explizite Auffaltung aller möglichen Exekutionsfolgen von Aktivitäten, wenn der entsprechende (stochastische) Zustandsgraph (ZG) erzeugt wird. Dies kann folglich zu einem exponentiellen Anwachsen des ZG in der Anzahl der Systemzustände führen. Dieses Phänomen wird hinlänglich als Zustandsraumexplosionsproblem bezeichnet. Wie unten dargestellt, behindert es oft nicht nur eine Zustandsgraph-basierte Analyse von Systemen, sondern macht diese schier unmöglich.

Traditionelle Vorgehensweise zur ZG-Exploration

Die traditionelle Technik zur Erzeugung aller erreichbaren Zustände eines modellierten Systems, nennt man *exhaustive* Zustandsraumexploration, wobei im Folgendem das Besuchen einzelner Zustände als *explizite* Exploration bezeichnet werden soll. Die Zuständsmenge, deren Elemente man vom initialen Zustand ausgehend besuchen kann, nennt man die Menge der erreichbaren Zustände. Die Datenstrukturen, die man benötigt, um alle erreichbaren Zustände zu generieren sind ein Pufferspeicher (Zustandspuffer) und ein strukturierter Speicherbereich (Zustandstabelle). Letzterer speichert die entdeckten Zustände und ersterer die entdeckten aber noch nicht explorierten. Ein Zustand wird als exploriert betrachtet, wenn alle seine Folgezustände bestimmt wurden. Da die Zugriffe auf den Zustandspuffer strukturiert erfolgen, können die derzeit nicht benutzten Einträge (Zustände) auf Sekundärspeicher ausgelagert werden. Somit und im Gegensatz zur Zustandstabelle, ist der Zustandspuffer nicht der Flaschenhals der exhaustiven und expliziten Zustandsraumexploration. Die Zustandstabelle dient als Datenbank, deren Sinn darin besteht zu entscheiden, ob ein Zustand erstmalig erreicht wurde oder nicht. Da die Zugriffe auf die Zustandstabelle unstrukturiert erfolgen, und Plattenzugriffe rechenaufwändig sind, ist die Anzahl der zu explorierenden Zustände durch den verfügbaren RAM-Speicher beschränkt. Als Konsequenz ist die Größe und Komplexität der zu analysierenden Systeme in der Praxis stark limitiert.
Zur Veranschaulichung denke man bspw. an einen ZG, dessen Zustandsdeskriptoren aus jeweils 10^3 Zählern bestehen, wobei ein jeder max. den Wert 256 annehmen kann. Somit benötigt die Speicherung eines einzigen Zustands ca. $0.977\,KByte$ und man braucht somit für 10^6 verschiedene Zustände schon ca. $0.93\,GByte$ an RAM-Speicher. Aber nicht genug, das Auffinden und evtl. Abspeichern von Zuständen, wobei letzteres auch die Auflösung von Hashkollisionen beinhaltet, induziert außerdem einen nicht ignorierbaren Laufzeitaufwand. Zur Veranschaulichung nehme man einmal an, dass es genug Speicher gibt, sagen wir für 10^8 Zustände. Bei einer durchschnittlichen Verarbeitungszeit von $1.3 \cdot 10^{-4}\,Sek.$[1] pro Zustand, muss man demnach ca. $3.6\,Std.$ warten, bis der ZG eines hochsprachlichen Modells exploriert wurde.

[1] Dies ist die durchschnittl. CPU-Zeit wie sie bspw. durch das SPN-basierte Tool DSPNexpress, verbraucht wurde um ca. 10^6 Systemzustände zu erzeugen und abzuspeichern, wobei eine 64-bit AMD Opteron Architektur verwendet wurde.

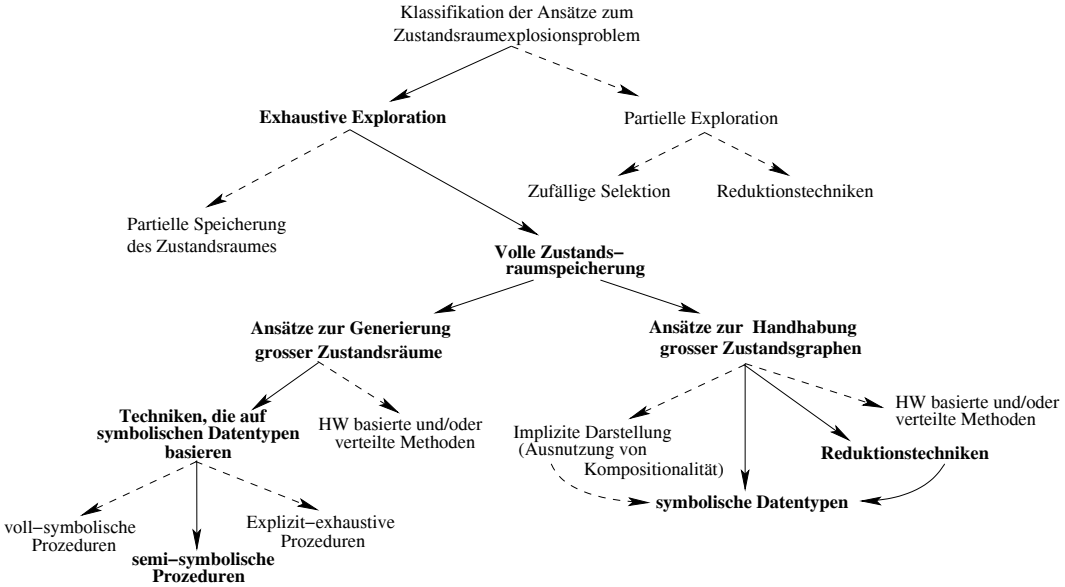


Abb. 3.1: Klassifizerungsschema der Ansätze zum Zustandsraumexplosionsproblem

Traditionelle Technik zur Speicherungs des ZG

Sind alle Zustände und alle Transitionen zwischen diesen erzeugt, werden auf Basis der Transitionsratenmatrix die Zustandswahrscheinlichkeiten numerisch berechnet. Jedoch kann die Grösse und Struktur des stochastischen ZG, dessen Transitionsratenmatrix im sog. "sparse matrix" Format abgespeichert wird, einen nicht ignorierbaren Speicherbedarf an den Tag legen. So verbraucht bspw. der ZG des bekannten "Kanban Manufacturing" Systems und des "Flexible Manufacturing" Systems (siehe engl. Teil: Kapitel 5) bei $\sim 2.5E6$ und $\sim 4.5E6$ Systemzuständen schon ~ 380 und $\sim 500\ MBytes$ zur Abspeicherung der jeweiligen Transitionsratenmatrix im "sparse matrix" Format.[2]

Aus diesen Gründen sind derzeitige Systemanalysen unter Verwendung der traditionellen Techniken auf Modelle mit deutlich weniger als 10^7 Systemzuständen beschränkt.

Klassifikation der Ansätze

Um das Problem, das sich aus der Zustandsraumexplosion auf der einen Seite und aus der Beschränkung der Verfügbarkeit von Speicher auf der anderen Seite ergibt, in den Griff zu bekommen, sind verschiedenste Ansätze entwickelt worden. Eine Klassifikation existierender Ansätze ist in Abb. 3.1 dargestellt, wobei wir uns auf Ansätze, die im Umfeld von hochsprachlichen Markov Modellen entwickelt wurden, konzentrieren. Auf der obersten Ebene kann man zwischen Ansätzen unterscheiden, die eine partielle und eine exhaustive Zustandsraumsuche durchführen. Die Methoden der ersten Klasse sind die sog. partiellen Analysemethoden. Es gibt verschiedene Möglichkeiten eine partielle Analyse zu organisieren. Die zwei bekanntesten Vertreter dieser Klasse sind die Reduktionstechniken und die probabilistischen Methoden. Die Reduktionstechniken zielen darauf ab, redundante Aktivitätssequenzen nicht zu explorieren. Dies kann bspw. durch Ausnutzung einer Äquivalenzrelation, definiert auf dem Systemverhalten, erzielt werden (siehe bswp. [God95]) oder bspw. durch Ausnutzung von benutzer-definierten Modellsymmetrien zur Anwendung des "state lumping" Theorems "on-the-fly" (siehe Abs. 2.4 (S. 24ff) des engl. Teils). Probabilistische Methoden erlauben die Speicherung große Zustandsräume. Bedingt dadurch, dass Hashkollisionen nicht aufgelösst

[2] Die Wert, die hier angegeben sind, erhält man, wenn man diese bekannten "benchmark" Modelle mittels des Möbius Modellierungswerkzeuges [DCC+02] bzw. mittels dessen Standardmoduls zur Analyse von Markov Reward Modellen untersucht, Details folgen im Kapitel 5 des engl. Teils, siehe hierzu Tabelle 5.1 (S. 121) und Tabelle 5.17.B (S. 148) im engl. Teil der Arbeit.

und somit verschiedene Zustände fälschlicherweise als identisch betrachtet werden, ergibte es sich, dass u.U. nur ein Teil aller Zustände erzeugt wird. Daraus ergibt sich, dass evtl. Zustände vergessen werden und die Wahrscheinlichkeit eines falschen Analyseergebnisses somit größer als Null ist.[3]

Die Ansätze, die eine exhaustive Zustandsraumsuche durchführen, kann man in zwei Klassen teilen:

(1) Ansätze, die einen reduzierten ZG speichern: Dies wird dadurch erreicht, dass man während der Exploration nicht benötigte Zustände nicht (permanent) speichert, z.B. durch Eliminierung von verschwindenden Zuständen “on-the-fly”.

(2) Bzgl. der exhautiven Ansätze, die den kompletten ZG speichern, kann man nun zwischen Methoden unterscheiden, die die Generierung sehr grosser Zustandsräume ermöglichen und zwischen Methoden, die auf eine Speicherung sehr grosser Zustandsgraphen, als auch auf die Berechnung von deren Perfomabilitätsmaße abzielen:

   (2.a) Ansätze zur Generierung sehr großer Zustandsgraphen: Eine exhaustive Exploration des Zustandsraumes kann effizient durch

      i. Verwendung leistungsfähiger HW, d.h. durch Verwendung von Massenspeicher und/oder verteilter HW (siehe bspw. [Kno99, HW06]), oder durch

      ii. Verwendung symbolischer Methoden (siehe deut. Teil: Abs. 3.3) erzielt werden.

   (2.b) Die Klasse der Ansätze zur Speicherung und Handhabung sehr grosser Zustandsgraphen kann wie folgt weiter unterteilt werden:

      i. Methoden die leistungsfähige HW und/oder verteilte HW verwenden (siehe bspw. [Kno99, HBB99, Meh04]).

      ii. Verfahren die den Zustandsgraphen aposteriori zur Generierung, durch Auffinden und Ausnutzung einer Äquivalenzrelation, reduzieren (siehe engl. Teil: Abs. 2.2.3, S. 17ff).

      iii. Methoden, die durch Verwendung von Kronecker-Operatoren eine implizite Repräsentation des ZG des Gesamtmodells erreichen (siehe bspw. [Pla85, Buc91, Sie95, CT96]), und/oder

      iv. Verfahren, die symbolische Datentypen (Entscheidungsdiagramme) verwenden (siehe deut. Teil: Abs. 3.3)

Wie in Abb. 3.1 dargestellt haben die symbolischen Methoden das besondere Merkmal, dass sie nicht nur die Generierung von sehr großen ZG unterstützen, sondern auch eine effiziente Abspeicherung derselbigen, d.h. vielmehr der zugehörigen Transitionsratenmatrix, erlauben. Darüber hinaus werden sie ausserdem auch im Rahmen anderer Methoden eingesetzt, um deren Effizenz zu steigern, als Bsp. seien die impliziten Matrixdarstellungstechniken [Sie98, CM99b] und die Reduktionsverfahren auf Basis von Entscheidungsdiagrammen [Sie02] unter anderem erwähnt. Im Sinne der obigen Klassifizierung ist der Beitrag dieser Arbeit zur Verminderung des Zustandsraumexplosionsproblem auf Basis eines (neuen) symbolischen Datentyps somit dreigeteilt: (a) Es wird eine Methode zur effizienten Generierung einer symbolischen Darstellung des ZG eines hochsprachlichen Markov Reward Modells erarbeitet. (b) Ein Ansatz zur effizienten Speicherung und Handhabung des erzeugten ZG wird eingeführt, so dass man die Performabilitätsmaße des zu untersuchenden Systems effizient berechnen kann. (c) Es wird ein Ansatz vorgestellt, der im Falle von benutzer-definierten Modellsymmetrien eine Reduktion des ZG realisiert, so dass die numerische Analyse mit

[3] Ein Überblick über die probabilistischen Methoden kann in [KL04] gefunden werden.

einer geringeren Anzahl von Systemzuständen arbeiten muss. Somit lässt sich das symbolische Rahmenwerk, das im Verlauf dieser Arbeit entwickelt werden wird, anhand der fett gedruckten Begriffe gemäß der Klassifizierung aus Abb. 3.1 charakterisieren.

3.3 Stand der symbolischen Techniken

Entscheidungsdiagramme ermöglichen die effiziente Speicherung von Funktionstafeln. Somit ist es nahe liegend, sie auch zur Darstellung der charackteristischen Funktion von endlichen Mengen und somit zur Repräsentation von Zuständsmengen und/oder Transitionsrelationen zu benutzen. Ansätze, die solche Methoden der Speicherung benutzen, werden gemeinhin als symbolische Repräsentationstechniken bezeichnet.

Symbolische Datentypen

Heutzutage ist der Einsatz von Binäre Entscheidungsdiagrammen (BDDs) in CAD-Werkzeugen Stand der Technik, da man mit ihnen i. Allg. Boolesche Funktionen besonders effizient darstellen kann. Darüber hinaus sind effiziente Algorithmen zu ihrer Manipulation bekannt [Bry86]. Innerhalb der letzten Dekade sind viele abgeleitete Typen entstanden, um Entscheidungsdiagramme nicht nur innerhalb der HW-Verifikation einzusetzen, sondern auch in anderen Bereichen, in denen man es mit große Mengen an Zahlensequenzen und deren Vorhaltung im RAM-Speicher zu tun hat, siehe bspw. [SS03]. Es ist deshalb nicht überraschend, dass auch der Bereich der stochastischen Modellierung Vorteil aus der symbolischen Darstellung von Mengen und Transitionsrelationen gezogen hat. Im Kontext der stochastischen Modellierung sind die bekanntesten Typen von Entscheidungsdiagrammen die *Multi-terminalen* - oder *algebraischen BDDs* (ADDs) [FM97], die *"multi-valued" Entscheidungsdiagramme* (MDDs) [KVBSV98] und *Matrixdiagramme* (MxD) [Min01].

Da ADDs die multi-terminale Erweiterung der BDDs darstellen, sind die wichtigsten BDD-Algorithmen i. Allg. direkt auf sie anwendbar und viele verschiedene Implementierungen existieren. Deswegen ist es auch nicht verwunderlich, dass gerade dieser Typ von Entscheidungsdiagramm eine lange Historie hat, wenn es um die Modellierung von Systemen geht. Jedoch hat sich gezeigt, dass im Kontext der hochsprachlichen Modellenbeschreibungen die BDD-spezifische "*don't-care*" Reduktionsregel i.d.R. von untergeordneter Bedeutung für die Speichereffizienz ist (zumindest für die Pfade, die zum terminalen 1-Knoten führen) [Par02].

Generierungstechniken

Techniken, die zur Erzeugung von symbolische Darstellung der ZG eines hochsprachlichen Modells verwendet werden, reichen von der expliziten Generierung aller Zustände (exhaustiv) [Web02, DKK02], bis zu den voll-symbolischen Techniken [PC98, KS02, AKN+00]. Bei letzteren werden die symbolische Darstellung sogar direkt aus der hochsprachlichen Modellbeschreibung abgeleitet, so dass sich Methoden dieser Klasse durch eine besondere Effizienz auszeichnen. Jedoch, im Gegensatz zu den Methoden, die eine konventionelle ZG-Exploration benutzen, verlangen die voll-symbolischen Techniken, dass die hochsprachliche Modellbeschreibungsmethode eine symbolische Semantik besitzt. Eine andere wichtige Klasse, deren Verfahren im Gegensatz zu den voll-symbolischen Verfahren unabhängig vom benutzten Modellbeschreibungsformalismus sind, ist die Klasse der so genannten semi-symbolischen Methoden [Sie98, HMKS99, Sie01, CM99b, CLS01]. Die Bezeichnung semi-symbolisch charakterisiert eine Kombination expliziter Exploration und rein symbolischer Manipulation, wobei im Gegensatz zu den explizit exhaustiven Methoden ein Kompositionsschema benutzt wird.[4] Kompositionalität scheint in diesem Zusammenhang maßgeblich zu sein, nicht nur für die semi-symbolischen Methoden, denn es reduziert nicht nur die Laufzeit, da nicht

[4] Die Anwendung eines Kompositionsschemas bedeutet, dass der ZG des Gesamtmodelles aus kleineren Komponenten, gewöhnlich aus den ZG der vom Benutzer spezifizierten Submodelle (submodell-lokalen ZG), aufgebaut wird. Details werden im engl. Teil, Kapitel 2 erörtert.

alle Aktivitätssequenzen explizit aufgefaltet werden müssen, sondern induziert auch eine gewisse Regularität und verringert somit den Spitzenspeicherverbrauch.

Der Einsatz der oben diskutierten ZG-Generierungstechniken ist jedoch auf folgende Fälle beschränkt:

(1) der hochsprachliche Formalismus ist von einer bestimmten Art [PC98, KS02, AKN+00],

(2) Beschränktheit der Komponenten des Zustandsdeskriptors, welcher entweder direkt im hochsprachlichen Modell spezifiziert ist [KS02, AKN+00] oder mittels einer Invariantenanalyse berechnet werden kann [PRCB94, DKK02].

(3) Die hochsprachliche Modellbeschreibung besitzt eine kompositionelle oder bestimmte dekomponierbare Struktur und die ZG der Submodelle können in Isolation generiert werden [CM99b, CLS01, HMKS99, AKN+00, Sie01, KS02, LS02].

Kürzlich entwickelte semi-symbolische, kompositionelle Methoden, wie in [CMS03, DKS03] dargestellt, erzeugen die submodell-lokalen ZG in einem Submodell-interdependenten Verfahren, um die oben aufgezählten Einschränkungen zu überwinden. Aus folgenden Gründen ist ihr Einsatz nichtsdestotrotz immer noch beschränkt:

(1) Die Technik, die in [CMS03] angewandt wird benötigt eine Dekomposition des hochsprachlichen Modells in unabhängige Partitionen, so dass der ZG des Gesamtmodells mittels eines Kronecker-Operator-getriebenen Kompositionsschemas erzeugt werden kann. (siehe engl. Teil: Abs. 2.5, S. 25ff).

(2) Die Technik, die in [DKS03] angewandt wird, verwendet die benutzerdefinierte modulare Struktur des Gesamtmodells, so dass Ineffizienzen auftreten, wenn die Interaktionen zwischen den Submodellen nicht limitiert ist.

(3) Nebenläufigkeit innerhalb der Submodelle wird i. Allg. nicht ausgenutzt. Folglich werden die verschränkten Sequenzen unabhängiger Aktivitäten auf Ebene der Submodelle voll expandiert, so dass im Falle von Submodellen mit extrem großen ZG die Methoden i.d.R. nicht effizient sind.

Die obige Diskussion ergibt folgende zentrale Anforderungen an einen neuen Ansatz:

- Die individuelle Behandlung von Zuständen (Exploration und Kodierung) sollte soweit wie möglich vermieden werden.
- Der Ansatz sollte sich auf kompositionelle und nicht-kompositionelle Modellwelten anwenden lassen, d.h. auf Modelle die modular oder nicht modular (monolithisch) strukturiert sind.
- Jedoch sollte der Ansatz auf Ebene des ZG irgendeine Form von Kompositionalität aufweisen.

ZG-Reduktionstechniken
Die Anzahl der Systemzustände, für die eine Wahrscheinlichkeit berechnet werden muss, ist auch unter den symbolischen Verfahren der Engpass der ZG-basierten Analyse von hochsprachlichen Markov Reward Modellen. Unter Umständen ist es möglich den ZG zu reduzieren, so dass für eine kleinere Anzahl von Systemzuständen Performabilitätsmaße berechnet werden müssen. Folgende Klassen von Ansätzen sind zu unterscheiden:

(1) Transformation des hochsprachlichen Modells:
Durch Anwendung von Transformationsregeln, die abhängig vom verwendeten hochsprachlichen Formalismus sind, ist es u.U. möglich das Ausgangsmodell in ein einfacheres umzuwandeln, welches die gleichen zeitl. Eigenschaften bzgl. der zu berechnenden Performabilitätsmaße zeigt. Das vereinfachte Modell kann dann, im Vergleich zum Ausgangsmodell, zu einem reduzierten ZG führen.

(2) "On-the-fly" Strategie:
Diese Techniken generieren von vornherein, also während der ZG-Exploration, einen reduzierten ZG und erlauben so die Analyse von Systemen, die sich sonst einer solchen entziehen.

(3) "A-posteriori" zur ZG-Exploration:
Diese Techniken generieren zunächst den kompletten, also nicht reduzierten ZG. Nun kann durch Anwendung bekannter Verfahren eine ZG-Reduktion erfolgen. Deshalb sind die Ansätze dieser Klasse auch allgemeiner als die "on-the-fly"-Strategien.

Aus den oben genannten Gründen ist die Entwicklung von ZG-Reduktionstechniken immer noch ein belebtes Forschungsfeld und hat in den letzten Jahren auch entsprechende symbolische Ansätze hervor gebracht. Jedoch ist bekannt, dass die bisher entwickelten symbolischen Verfahren, wie ihre expliziten Gegenstücke, nicht sonderlich effizient sind.

Numerische Lösungstechniken

Ist eine symbolische Darstellung des ZG eines hochsprachlichen Modells erst einmal erzeugt, so folgt als nächstes die Bestimmung der Zustandswahrscheinlichkeiten. Jedoch erscheinen im Kontext symbolischer ZG-Repräsentation nur der Einsatz iterativer, numerischer Lösungsverfahren sinnvoll. Wie aus der Praxis bekannt, sind die voll symbolischen Ansätze, die neben einer symbolisch repräsentierten Transitionsratenmatrix auch symbolisch repräsentierte Iterationsvektoren verwenden, nicht sehr effizient. Deswegen hat sich derzeit in der Praxis die sog. hybride Lösungstechnik [Par02] weit verbreitet. Diese Verfahren benutzen symbolisch dargestellte Transitionsraten- oder Generatormatrizen, die Iterationsvektoren werden hingegen als Arrays gespeichert, was die einzelnen Iterationsschritte wesentlich beschleunigt. Sind die Zustandswahrscheinlichkeiten bekannt, so müssen im nächsten Schritt die benutzerdefinierten Performabilitätsmaße berechnet werden, wobei hierfür die sog. Raten- und Impuls-Rewards bestimmt und entsprechend aggregiert werden müssen.

3.4 Eigener Beitrag

In dieser Arbeit wird ein symbolischer Rahmen zur Analyse sehr großer, (finiter) Markov Reward Modelle mit kontinuierlicher Zeit vorgestellt. Zur Repräsentation der Markov Reward Modelle wird ein neuer Typ von Entscheidungsdiagramm vorgestellt, welchen wir als "zero-suppressed" multi-terminale binäre Entscheidungsdiagramme (ZDD) bezeichnen. Um mit ZDDs umgehen zu können, werden außerdem die entsprechenden symbolische Algorithmen entwickelt. Dieser neue Typ einer symbolischen Datenstruktur wird innerhalb eines neuen, semi-symbolischen Verfahrens zur effizienten Generierung einer symbolischen Repräsentation eines Markov Reward Modells, wie es sich aus einer hochsprachlichen Beschreibungen ableiten lässt, benutzt. Dieses Schema, welches wir als Aktivitäts/Reward-lokales Schema bezeichnen, basiert auf einer partiellen expliziten ZG-Exploration, einem neuen symbolischen Kompositionsschema und einem neuen Algorithmus zur symbolischen Erreichbarkeitsanalyse. Im Gegensatz zu bekannten Verfahren benutzt es Modell-inhärente Strukturen und keine explizit vom Benutzer spezifizierten. Folglich ist das Aktivitäts/Reward-lokale Schema nicht nur für kompositionelle Modellwelten geeignet, sondern in besonderem Maße für monolithische, d.h. also auch in den Situationen in denen Modelle weder in eine Kronecker-Operatorkonforme Struktur dekomponiert werden (siehe Abs. 2.5.6, S. 29f). können und/oder die Submodell-lokalen ZG in disproportionalen Größen vorliegen.

Um den ZG, dessen Zustandswahrscheinlichkeiten berechnet werden müssen, zu reduzieren, erweitern wir das Aktivitäts/Reward-lokale Schema um einen symbolischen Algorithmus, der eine ZG-Reduktion durchführt, vorausgesetzt, dass benutzerdefinierte Symmetrien vorliegen.

Zur Berechnung der Zustandswahrscheinlichkeiten erweitern wir das hybride Lösungsverfahren, so dass es auf die hier entwickelten ZDDs anwendbar ist. Da wir auch benutzerdefinierte Performabilitätsmaße behandeln, ergeben die einzelnen Beiträge dieser Arbeit einen vollständigen Ansatz zur ZG-basierte Analyse von hochsprachlichen Markov Reward Modellen mit sehr großen Zustands/Transitionssystemen. Um die Anwendbarkeit der entwickelten Konzepte zu bewerten, wurde eine Implementierung innerhalb des Multi-Formalismus-Leistungsanalyse-Werkzeugs Möbius [DCC+02] realisiert. Diese erlaubt nicht nur die Analyse bekannter Benchmark-Modelle, sondern auch die Bewertung der Verfügbarkeit eines “Adjunkten Prozessor-Systems”, wie es in der Telekommunikationsindustrie Verwendung findet, um die Leistungsfähigkeit des hier präsentierten Ansatzes zu untersuchen.

3.5 Aufbau der Arbeit

Der engl. Teil dieser Arbeit ist wie folgt aufgebaut: Kapitel 2 wiederholt benötigtes, theoretisches Hintergrundwissen und führt entsprechende Definitionen ein (siehe hierzu auch Anhang A). Kapitel 3 präsentiert den neuen Datentyp, und führt die wichtigsten Algorithmen zu seiner effizienten Manipulation ein. Ein Überblick über ZDD-basierte Repräsentation von Mengen und Matrizen, sowie eine Einführung in die hybride Lösungsmethode zum Lösen von linearen und differentialen Gleichungssystemen schließt das Kapitel ab. Kapitel 4 erklärt unseren neuen Ansatz zur effizienten Erzeugung von symbolischen Repräsentationen von Markov Reward Modellen aus ihren hochsprachlichen Spezifikationen. Empirische Ergebnisse, einschließlich verschiedener Vergleiche zwischen anderen Werkzeugen und unserer Implementierung, werden in Kapitel 5 präsentiert. Kapitel 6 schließt diese Arbeit ab, indem ein Zusammenfassung und ein Überblick über zukünftige Arbeiten gegeben wird.

4

Übersetzung des Kapitels “Conclusion” (Schlussbetrachtung)

4.1 Zusammenfassung

Diese Arbeit führt nicht nur einen neuen Typ von Entscheidungsdiagramm ein, sondern entwickelt auch einen neuen semi-symbolischen Ansatz zur Analyse von hochsprachlichen Markov Reward Modellen mit sehr großen Zustandsgraphen. Insgesamt trägt diese Arbeit somit zur Abschwächung des Zustandsraumexplosionsproblems, wie es sich im Kontext hochsprachlicher Markov Reward Modelle manifestiert, bei. Im Folgenden sollen einige Aspekte der präsentierten Arbeit kurz rekapituliert werden.

4.1.1 Zero-suppressed multi-terminale BDDs

Boolesche Funktionen mit kleinen Erfüllbarkeitsmengen, bei denen die erfüllenden Belegungen auch noch viele mit 0 besetzte Positionen beinhalten, führen u.U. zu sehr großen BDD-basierten Darstellungen. In solch einem Szenario hat sich gezeigt, dass sog. *zero-suppressed* BDDs (z-BDDs) [Min93] hilfreich sind. In dieser Arbeit werden z-BDDs um den multi-terminalen Fall erweitert, um so effizient Pseudo-Boolesche Funktionen und charakteristische Funktionen stochastischer Transitionsrelationen darzustellen.

Partiell gemeinsame ZDDs (pZDDs)

Um eine Pseudo-Boolesche Funktion aus einem ZDD-Graphen korrekt abzuleiten, muss die Menge der Funktionsvariablen der repräsentierten Funktion bekannt sein. Als Konsequenz ergibt sich, dass ein ZDD-Knoten (und der darin verwurzelte Graph), ohne zusätzlich mit Funktionsvariablen annotiert zu sein, nicht eindeutig eine Pseudo-Boolesche Funktion repräsentiert. Dies liegt darin, dass übersprungene Variablen entweder als nicht maßgeblich oder als zero-suppressed interpretiert werden können. Dies ist insbesondere dann problematisch, wenn man in einer gemeinsamen BDD-Umgebung operiert, da hier Eindeutigkeit nicht nur eine Grundbedingung, sondern auch in Bezug auf Effizienz unverzichtbar ist. Um dieses Problem anzugehen entwickelt diese Arbeit das Konzept der partiell gemeinsamen ZDDs (pZDDs), das verlangt, dass jeder Knoten mit einer Menge an Funktionsvariablen ausgestattet wird, so dass (schwache) Kanonizität garantiert ist.

Algorithmen zum Umgang mit pZDDs

Dieser neue Typ von Entscheidungsdiagramm verlangt ein Re-Design bekannter BDD-Algorithmen [Bry86], wobei diese Arbeit die entsprechenden Varianten entwickelt, wie bspw. den p`ZApply-`, den p`ZAnd-`, den p`ZRestrict-` und p`ZAbstract`-Algorithmus. Im Gegensatz zum theoretischen Konzept der pZDDs, sind diese Algorithmen so beschaffen, dass sie die Implementierung von pZDDs innerhalb gewöhnlicher BDD-Umgebung erlauben, obwohl in diesen die einzelnen DD-Knoten i. Allg. nicht mit Funktionsvariablen ausgestattet werden können. Diese Strategie hat folgende wesentliche Vorteile: (a) man muss nicht die einzelnen Mengen der Funktionsvariablen für jeden Knoten abspeichern und (b) die gemeinsame Verwendung von pZDD-Graphen erhöht wird. Jedoch hat eine solche Vorgehensweise auch den Nachteil, dass die Semantik eines Knotens nicht mehr eindeutig ist. Dies ist insbesondere dann problematisch, wenn es um das Testen von Äquivalenz geht. Im Falle gewöhnlicher BDD-Umgebungen muss man dafür nur die Speicheradressen der Wurzelknoten vergleichen. Dies ist von besonderer Bedeutung, da der Test auf Äquivalenz unzählige Male bei der Manipulation von ZDDs auftritt, nicht nur um die terminalen Rekusionabbruchsbedingungen

zu testen, sondern auch wenn es um das Auffinden von bereits bekannten Ergebnissen in der Vorberechnungstabelle (pre-computed table) geht. Um dieses Problem zu lösen wird folgende Strategie verfolgt: Wenn man auf dem Graphen eines pZDDs Z operiert, übergibt man einfach die Menge von Zs Funktionsvariablen ($\mathcal{V}^{\mathsf{Z}}$) als zusätzliches Argument an den, den Graphen manipulierenden Algorithmus. Jedes Mal, wenn ein Test auf Äquivalenz benötigt wird, braucht man nur die Speicheradressen der pZDD-Knoten und die Mengen der Funktionsvariablen zu vergleichen. Sind die jeweiligen Paare identisch, so sind es auch die repräsentierten Funktionen.

4.1.2 Aktivitäts/Reward-lokales Schema

Wenn der Formalismus eines hochsprachlichen Modells keine symbolische Semantik hat, ist der einzige Weg eine symbolische Darstellung des zugrunde liegenden Markov Reward Modells zu erhalten, indem man das Modell explizit ausführt. Dabei wird die iterativ Ausführung und die Erzeugung der entsprechenden symbolischen Strukturen solange fortgesetzt, bis ein globaler Fixpunkt erreicht ist. Wird dies jedoch für alle Zustände und alle Transitionen explizit durchgeführt, verhindern der Zeitbedarf und der Spitzenspeicherverbrauch die Anwendung einer symbolischen Zustandsgraphen-Repräsentationstechnik in der Praxis.

Symbolische Zustandsgraphen-Generierung

Um effizient eine symbolische Darstellung einer kontinuierlichen Markovkette eines hochsprachlichen Modells zu erzeugen, nutzt das aktivitäts-lokale Schema, das in dieser Arbeit entwickelt wird, nur lokale Information. D.h. die Grundidee ist eine Partitionierung des zu generierenden Zustandsgraphen in Mengen von Transitionen, die die gleiche Aktionsbeschriftung führen, wobei die Quell- und Zielzustände auf die Positionen reduziert werden, von denen die transitions-induzierende Aktivität abhängt. Dies ergibt ein Transitionssystem für jede Aktivität, die wir deshalb als aktivitäts-lokale Transitionssysteme bezeichnen. Die vollständige Transitionsrelation des hochsprachlichen Modells wird dann durch Anwendung eines symbolischen Kompositionsschemas auf Basis der aktivitäts-lokalen Transitionssysteme erzeugt. Die explizite Explorierung der aktivitäts-lokalen Transitionssysteme folgt hierbei einer selektiven Breitensuche. Eine solche Suchstrategie erhält man, wenn man eine Abhängikeitsrelation bei der Auswahl der zu betrachtenden Aktivitäten beachtet und in jedem neu besuchten Zustand nur versucht jene Aktivitäten auszuführen, die in diesem Zustand noch nicht getestet wurden. Jedoch im Gegensatz zu Standardverfahren, werden nicht volle Zustandsdeskriptoren bei diesem Test betrachtet, sondern nur die Werte, die sich auf Positionen beziehen, von denen die jeweilige Aktivität abhängt. Darüber hinaus testet man auch nicht alle Aktivitäten, sondern lediglich jene, die von der zuletzt ausgeführten Aktivität bzgl. des momentanen Zustandes abhängen. Als Konsequenz ergibt sich somit ein explizites Zustandsgraphen-Generierungs- und Zustandsgraphen-Kodierungsverfahren für das aktivitäts-lokale Schema, das in der Praxis i.d.R. partieller Natur ist. Wird ein lokaler Fixpunkt erreicht, d.h. sind für eine gegebene Menge von Zuständen alle Sequenzen anhängiger Aktivitäten erzeugt, so wird ein symbolisches Kompositionsschema angewendet. Die resultierende symbolische Struktur repräsentiert jedoch das potentielle Transitionssystem, so dass an dieser Stelle eine symbolische Erreichbarkeitsanalyse ausgeführt werden muss. Um die symbolische Erreichbarkeitsanalyse zu beschleunigen, entwickelt diese Arbeit eine verbesserte Version, die nach einer “Quasi”-Tiefensuche vorgeht und mit den aktivitäts-lokalen Entscheidungsdiagrammen arbeitet, d.h. auf einer partitionierten Transitionsrelation. Nach Terminierung der symbolische Erreichbarkeitsanalyse hat man nun die Menge der bisher erreichbaren Zustände erzeugt. Da dies allerdings in Zustände münden kann, die neues explizites Modellverhalten initiieren könnten, wird eine Re-Initialisierungsroutine benötigt. Existieren solche Zustände, initiiert die Routine eine neue Runde expliziter Zustandsgraphen-Exploration, -Kodierung, symbolischer Komposition und symbolischer Erreichbarkeitsanalyse. Hierbei können mehrere Runden von Nöten sein bis ein globaler Fixpunkt erreicht

wird und eine vollständige Repräsentation der, dem hochsprachlichen Modell immanenten zeit-kontinuierlichen Markovkette konstruiert wurde.

Erzeugung symbolischer Repräsentationen von Rewardfunktionen
Um die Performabilitätsmaße des zu analysierenden Systems zu definieren, ist es dem Benutzer möglich sog. Performanzvariablen auf Ebene der hochsprachlichen Modellbeschreibung zu spezifizieren. Traditionell berechnet man die zustands- und aktivitäts-abhängigen Performanzvariablen während der Generierung des Zustandsgraphen. Das Aktivitäts/Reward-lokale Schema besucht während der Zustandsgraphen-Generierung u.U. nur einen Teil der Zustände explizit. Unter der Voraussetzung, dass die Rewardfunktionen von beliebiger Komplexität sein können, aber nur von einem Teil der Zustandvariablen abhängen, erscheint es somit sinnvoll, ihre symbolische Darstellung zu erstellen, wenn die symbolische Repräsentation des Zustandsgraphen vorliegt. Um nun so wenig Zustände als möglich explizit zu verarbeiten, verwendet diese Arbeit erneut nur lokale Informationen. D.h. nach Berechnung des Rewardwertes eines Zustandes, werden nur jene Positionen des Zustandsdeskriptors in der symbolischen Darstellung gespeichert, die sich auf die Variablen der Rewardfunktion beziehen. Diese Strategie definiert erneut eine Äquivalenzrelation auf der Menge der Systemzustände bzgl. der jeweiligen Rewardfunktion, so dass die Anzahl an Zuständen, die explizit betrachtet werden muss i. Allg. signifikant reduziert werden kann. Durch Aggregation der symbolisch dargestellten Rewardfunktionen, entsprechend der vom Benutzer gegebenen Definitionen, können symbolisch Darstellungen der Performanzvariablen erzeugt werden. Nun muss die Wahrscheinlichkeitsverteilung auf der Menge der Systemzustände berechnet werden, da diese in Kombination mit den symbolisch repräsentierten Performanzvariablen zu den Performabilitätsmaßen des zu analysierenden Systems aggregiert werden können.

4.1.3 Berechnung der Zustandswahrscheinlichkeiten

Um die Zustandswahrscheinlichkeiten zu berechnen, präsentiert diese Arbeit eine Variante der hybriden Lösungsmethode, wie sie für andere symbolische Datentypen bekannt ist. Bei diesem Ansatz wird die Generator-Matrix mittels eines Entscheidungsdiagramms dargestellt und die Iterationsvektoren als einfache Arrays des Datentyps “double” gespeichert. Wenn n Boolesche Variablen zur Kodierung eines Zustands benötigt werden, dann gibt es 2^n potentielle Zustände, von denen allerdings meist nur ein kleiner Teil erreichbar ist. Speichereinträge für unerreichbare Zustände sind eine Verschwendung von Arbeitsspeicher und würden die Analysierbarkeit von Systemen stark einschränken; zur Veranschaulichung denke man an 134 Millionen Zustände, für die man schon 1 GByte RAM-Speicher verbraucht, um die Zustandswahrscheinlichkeiten im Speicher vorzuhalten. Aus diesem Grund ist es nötig, eine dichte Nummerierung auf der Menge der erreichbaren Zustände zu realisieren. Dies kann mit dem Konzept des “offset-labeling” erzielt werden. Das “offset-labeling” von Knoten erlaubt es, den Zeilen- und Spaltenindex in dem dichten Nummerierungssystem eines Matrixelements zu berechnen, während man die symbolische Darstellung der Matrix traversiert.
Die Speichereffizienz der symbolischen Matrixrepräsentation geht zu Lasten des Rechenaufwand, der durch die rekursive Traversierung der symbolischen Struktur verursacht wird. Aus diesem Grund ersetzt man die unteren Ebenen der symbolischen Struktur durch sog. “sparse matrix” Darstellungen, was insbesondere gut für Block-strukturierte Matrizen funktioniert. Um das gute Konvergenzverhalten der Vorwärts- und Rückwärts-Gauss-Seidel Methode und ihrer relaxierten Varianten zu verwenden, ist ein reihen- oder spaltenweiser Zugriff auf die Matrixelemente nötig. Als Kompromiss implementierten wir die sog. Pseudo-Gauss-Seidel Methode, wie sie aus dem Bereich der algebraischen Entscheidungsdiagramme bekannt ist. Diese Methode partitioniert die Matrix in Blöcke, welche in einer geordneten Reihenfolge besucht werden. Innerhalb eines jeden Blocks erfolgt dann der Elementzugriff in beliebiger Reihenfolge. Insgesamt ergeben die oben dargestellten Verfahren nun ein Entscheidungsdiagramm, bei dem die unteren Ebenen und ggf. auch die oberen Ebenen in ein sog. “sparse matrix” Format gebracht werden. Unter Verwendung einer entsprechenden iterativen

Lösungsmethode können nun die Zustandswahrscheinlichkeiten berechnet und in einem Array abgespeichert werden.

4.1.4 Berechnung der Performabilitätsmaße

Hat man symbolische Darstellungen von Performanzvariablen und die Zustandswahrscheinlichkeiten gegeben, so ist man in der Lage, die Performabilitätsmaße des zu analysierenden Systems zu bestimmen. Dafür verwendet diese Arbeit einen neuartigen Algorithmus, der die Momente der Performanzvariablen mittels Graphtraversierung berechnet.

4.2 Vorteile von *p*ZDDs und dem Aktivitäts/Reward-lokalen Schema

Die Vorteile der Verwendung von *p*ZDDs können wie folgt zusammengefasst werden:

- Das Konzept der *p*ZDDs und die daraus resultierenden Algorithmen erlauben die effiziente Allokation und Manipulation von *z*-BDDs und deren multi-terminale Erweiterungen innerhalb der standardmäßigen BDD-Pakete, auch wenn die Entscheidungsdiagramme unterschiedliche Funktionsvariablen besitzen. Dies wurde nach unserem Kenntnisstand zuvor nicht wissenschaftlich untersucht.

- Im Zusammenhang mit symbolischen Darstellungen von Zustandsmengen, wie sie im Bereich der hochsprachlichen Modelle auftreten, haben ZDDs gezeigt, dass sie der oft speichereffizienteste Datentyp sind. Dieser Vorteil kann auch im Zusammenhang der Darstellung von stochastischen Transitionssystemen beobachtet werden.

- Aus den Messungen dieser Arbeit ergab sich, dass ZDDs als performanter als ihre algebraischen Gegenstücke, die ADDs, einzuschätzen sind, zumindest wenn es um Generierung von Zustandsgraphen und die Anwendung der hybriden Lösungsmethode zur Lösung von Linearen- und Differential-Gleichungssystemen geht. Basierend auf ihrer höheren Kompaktheit, ist man in der Lage größere Anteile des Entscheidungsdiagramms in ein “sparse matrix” Format umzuwandeln und so weitere Geschwindigkeitsvorteile zu realisieren oder eine weitere Reduktion des Blockiterationsvektors im Falle der Pseudo-Gauss-Seidel Methode zu erzielen, so dass u.U. größere System untersucht werden können.

Die Vorteile des neuen semi-symbolischen Verfahrens zur Generierung einer symbolischen Darstellung des Transitionssystems eines hochsprachlichen Modells, können wie folgt zusammengefasst werden:

- I. Allg. muss nur ein kleiner Anteil der Transitionen der Markovkette explizit erzeugt und kodiert werden. Der Großteil wird auf Ebene der symbolischen Darstellung durch das symbolische Kompositionsschema erzeugt.

- Da der Ansatz kompositioneller Natur ist, ist im Gegensatz zu den monolithischen Verfahren nicht nur der Laufzeitbedarf moderat, sondern auch der Spitzenspeicherverbrauch.

- Die Erreichbarkeitsanalyse wird auf Ebene der symbolischen Datenstrukturen durchgeführt. Hier schlägt diese Arbeit ein neues “quasi”-tiefe-suchendes Verfahren vor, dass mit seiner Effizienz überzeugt.

- Zur Generierung der symbolischen Darstellungen von Rewardfunktionen müssen i. Allg. nur wenige Zustände explizit verarbeitet werden. Dies ist wesentlich effizienter als die traditionelle Herangehensweise, unter der für jeden Zustand der Rewardwert explizit berechnet und abgespeichert werden muss.

- Diese Arbeit schlägt die symbolische Darstellung benutzerdefinierter Performanzvariablen vor, wobei deren Momente mittels eines Graph-traversierenden Algorithmus berechnet werden können. Dies ist im Vergleich mit traditionellen Ansätzen speicher- und laufzeiteffizient.
- Der Aktivitäts/Reward-lokale Ansatz beruht auf einer expliziten Ausführung von Aktivitäts- und Rewardfunktionen. Somit ist seine Anwendung nicht auf eine bestimmte hochsprachliche Modellbeschreibungstechnik beschränkt.
- Das Modell wird automatisch auf Ebene der Aktivitäten partitioniert, d.h. die explizite Angabe einer Partitionierung durch den Benutzer ist nicht nötig.
- Das Schema verlangt keine besondere Modellstruktur. Es ist somit, im Gegensatz zu anderen bekannten Verfahren, nicht auf Modelle beschränkt, die eine Kroneckerprodukt-konforme Struktur haben.

4.3 Zukünftige Arbeiten

Zukünftige Schritte, insofern sie eine Verbesserung des in dieser Arbeit präsentierten Ansatzes oder der symbolischen Technik im Rahmen der Performabilitätsanalyse i. Allg. betreffen, handeln von verschiedenen Aspekten. Diese Aspekte sollen nun im einzelnen besprochen werden:

Verbesserung der Zustandsgraphen-Generierungsprozedur
Wie am TQN-Modell dargestellt (siehe engl. Teil: Abs. 5.2.1), existieren für das Aktivitäts/Reward-lokale Schema sog. “worst case” Szenarien. Ebenfalls erscheint die Performanz im Falle von Kroneckerprodukt-konformen Modellen, wie bspw. im Falle des Kanban-Modells, verbesserungswürdig. Ein genauerer Blick auf das hochsprachliche Modell zeigt, dass einige Aktivitäten in diesen Modellen extensiv ausgeführt werden. Andererseits ist bekannt, dass die Zustandsgraphen dieser Modelle effizient kompositionell aufgebaut werden können, wobei der *Sync*-Kompositionsoperator Verwendung findet. D.h. im Falle der Benutzer gibt eine entsprechende Dekomposition des Gesamtmodells an, oder das Modell ist entsprechend kompositionell aufgebaut, dann können die lokalen Zustandsgraphen der einzelnen Modellpartitionen mittels des aktivitäts-lokalen Schemas effizient erzeugt werden. Eine symbolische Darstellung des Gesamtsystems kann dann mittels Anwendung des *Sync*-Operators (siehe Gl. 2.9) erreicht werden, wobei dessen symbolische Variante bereits in [Sie02] eingeführt wurde.

Symbolische Ausführung von Aktivitäten
Das Aktivitäts/Reward-lokale Schema ist ein semi-symbolischer, kompositioneller Ansatz, um die symbolische Darstellung eines Markov Reward Modells zu erzeugen. Somit setzt es die Anwendung eines symbolischen Kompositionsschemas und einer symbolischen Erreichbarkeitsanalyse voraus, wobei die entsprechenden Schritte separat, unter massiver Verwendung von DD-Manipulationsalgorithmen stattfinden. Es scheint eine gute Idee zu sein, die Komposition und Ein-Schritt-Erreichbarkeit in einem einzigen DD-basierten Algorithmus zu integrieren. Dieser Algorithmus hat hierbei ein aktivitäts-lokales Transitionssystem und die Menge der bisher erreichten Zustände als Eingabeparameter. Er wird nun solange wiederholt, bis ein Fixpunkt erreicht ist und eine Repräsentation der Menge der erreichten Zustände erzeugt wurde. Insbesondere würde eine solche Vorgehensweise nicht die Anwendung eines Kompositionsschemas benötigen und würde verschiedene DD-Operationen in einem einzigen Algorithmus integrieren, so dass u.U. insgesamt ein besseres Cachingverhalten erzielt werden kann.

Dynamisch variierende hochsprachliche Modellbeschreibungen
Das Aktivitäts/Reward-lokale Schema, wie es in dieser Arbeit eingeführt wurde, ist so konstruiert, dass es von statisch strukturierten hochsprachlichen Modellen ausgeht. Eine interessante Klasse von hochsprachlichen Modellspezifiaktionsmethoden, die ebenfalls auf endliche Zustandsgraphen abgebildet werden, erlauben jedoch die Prozessallokation und Prozesslöschung zur Modellausführungszeit, siehe bspw. [CB06]. Das Aktivitäts/Reward-lokale Schema scheint aber so erweiterbar, dass es in einem solchen Kontext eingesetzt werden kann. Im Zusammenhang mit ZDDs, bei denen die Einführung neuer Variablen keinen signifikanten Mehraufwand bedeutet, sieht dies sehr vielversprechend aus.

Parallelisierung des Aktivitäts/Reward-lokalen Schemas
In [Wei05] sammelten wir erste Erfahrungen mit der Parallelisierung des Aktivitäts/Reward-lokalen Schemas. Wie sich zeigte, verlangsamen die Schreiboperationen auf den ZDDs unter gegenseitigem Ausschluß den Prozeß der symbolischen Zustandsgraphen-Generierung. Jedoch konnte der Autor von [Wei05] durch die Implementierung eines “Thread-Pools” auch andere vielversprechende Schritte des Aktivitäts/Reward-lokalen Schemas parallel ausführen. Dies scheint insbesondere dann von Vorteil zu sein, wenn ein Modell extensive Zustandsgraphen-Exploration verlangt. Da die symbolische Erreichbarkeitsanalyse auch weiterhin die Hauptursache des CPU-Rechenzeitverbrauchs ist, könnte jedoch eine weiterführende Untersuchung, wie man symbolische Algorithmen besser parallelisieren kann, Verbesserungen bringen.

Verbesserung der numerischen Analyse
Wie im Abs. 5.5 des engl. Teils heraus gestellt wurde, ist der limitierende Faktor der Zustandsgraphen-basierten Analyse von hochsprachlichen Markov Reward Modellen, die Anzahl der Zustände, für die Wahrscheinlichkeiten ausgerechnet werden müssen. Um die derzeitigen Methoden zu verbessern erscheint eine dreiteilige Strategie angebracht:

(1) Symbolische Bisimulations Algorithmen: In Abs. 4.6.1 des engl. Teils wurde ein symbolischer Algorithmus zur Berechnung einer reduzierten bisimularen CTMC angegeben. Jedoch ist dieser Ansatz darauf beschränkt, dass innerhalb des hochsprachlichen Modells benutzerdefinierte Symmetrien vorliegen. Im Falle, dass die Spezifikation eine solche explizite Angabe von Symmetrien nicht beinhaltet, können allgemeinere Algorithmen eingesetzt werden. Diese Algorithmen, deren symbolische Varianten in [Sie02] eingeführt werden, sind jedoch dafür bekannt, nicht sonderlich effizient zu sein. In [DKS05] wird ein Ansatz präsentiert, der die bisimularen Strukturen auf Ebene des symbolisch repräsentierten Zustandsgraphen auffindet. Die vorgeschlagene Prozedur sieht sehr vielversprechend aus, da Submodell-induzierte Symmetrien direkt auf der symbolischen Repräsentation des Zustandsgraphen aufgespürt werden. Eine Prozedur um Submodell-induzierte Symmetrien im Rahmen des Aktivitäts/Reward-lokalen Schemas zu erkennen scheint einfach realisierbar: Man muss lediglich die aktivitäts-lokalen Transitionssysteme für jedes Submodell separat komponieren und diese dann für die entsprechenden Submodelle vergleichen. So ein Vergleich könnte entweder mittels eines entsprechenden symbolischen Algorithmus durchgeführt werden, oder durch Umbenennung der ZDD-Variablen und einem anschließenden Test auf Äquivalenz. Als Ergebnis wüsste man sofort, ob Submodelle mit der gleichen Spezifikation auch wirklich das gleiche zeitl. Verhalten besitzen. Als wesentlicher Vorteil eines solchen Ansatzes wäre zu nennen, dass man auf den Einsatz eines *Rep*-Operators verzichten und den *Join*-Operator flexibler benutzen könnte, so dass beliebige kompositionelle Modellstrukturen spezifiziert, aber dennoch Submodell-induzierte Symmetrien ausgenutzt werden können [Oba98]. Um noch einen Schritt weiter zugehen, scheint es außerdem lohnenswert, die Effekte von Symmetrien auf die symbolische Repräsentation des Zustandsgraphen zu studieren. Diese effizient zu erkennen und auszunutzen, würde dann eine minimale Bisimulationsrelation ergeben, was im Falle des Ausnutzens von Submodell-induzierter Symmetrie nicht notwendigerweise der Fall sein muss.

(2) Approximative Lösungsmethoden: Der “multi-level” Ansatz [HL94] ist dafür bekannt, dass er auch effizient in den Situationen eingesetzt werden kann, in denen die Generatormatrizen der aggregierten Systeme symbolisch oder implizit dargestellt sind. Jedoch sind extensive Manipulationen großer symbolischer Datenstrukturen rechenaufwendig, deswegen ist klar, dass der “multi-level” Ansatz im Zusammenhang symbolischer Darstellungen nur dann effizient realisierbar ist, wenn die Transitionsratenmatrizen der verschiedenen Aggregationsstufen nicht jedes mal erzeugt werden, wenn die jeweilige Aggregationsstufe besucht wird [Buc06]. Jedoch benötigt auch der “multi-level” Ansatz die Allokation der Wahrscheinlichkeitsvektoren, so dass dadurch die Anzahl der Zustände beschränkt ist. Somit ist auch die Anwendung der “multi-level” Methode in der Praxis beschränkt.
In [BG05] präsentieren die Autoren einen Ansatz der durch den “multi-level” Ansatzes inspiriert zu sein scheint, jedoch ausgehend von einem Zustandsgraphen eines aggregierten Systems, so dass die Anzahl der Zustandswahrscheinlichkeiten, die berechnet werden müssen, deutlich reduziert ist. Ein approximatives Ergebnis wird dadurch erhalten, dass man die Aggregation der Zustände variiert, deren Wahrscheinlichkeit berechnet und die Transitionsraten zwischen den aggregierten Zuständen manipuliert. Diese Prozedur wird solange wiederholt, bis ein Fixpunkt erreicht wird. Es wäre nun von Interesse zu untersuchen, ob ein solches Vorgehen für alle hochsprachlichen Modellbeschreibungsmethoden funktioniert, ob die Aggregationen des Ausgangssystems beliebig gewählt werden können, und ob diese Methode effizient auf einer symbolischen Datenstruktur umgesetzt werden kann.

(3) Parallelisierung: Die Autoren von [BH01] stellen einen parallelisierten numerischen Löser vor, welcher die standardmäßigen “sparse matrix” Formate benutzt. Es gilt nun zu untersuchen, ob man in diesem Kontext nicht auch symbolische Datenstrukturen verwenden kann, wobei die Parallelisierung der Pseudo-Gauss-Seidel Methode sehr vielversprechend aussieht.