Symbolic Reachability Graph Generation for High-level Models based on Synthesized Operators

Kai Lampka
lampka@tik.ee.ethz.ch
Computer Engineering and Communication Networks Lab.,
ETH Zurich, Switzerland

4. August 2009

TIK-Report 307,
ftp://ftp.tik.ee.ethz.ch/pub/publications/TIK-Report-307.pdf

Abstract

This paper introduces a new technique for generating Binary Decision Diagrams (BDDs) representing high-level model's underlying state/transition systems. The obtained decision diagram may serve as input for various analysis methods such as symbolic (probabilistic) model checking and/or Markovian performance and reliability analysis. As usual the proposed technique makes use of partitioned symbolic reachability analysis. However, contrary to existing techniques it neither relies on pregenerated symbolic representations of transition relations, nor does it make use of standard BDD-manipulating algorithms. Instead, symbolic reachability analysis is carried out by means of customized BDD-algorithms directly synthesized from high-level models to be analyzed. Overall the presented approach yields the core of a new tool bench for the symbolic analysis of state-based system descriptions. The tool bench is implemented on top of the Eclipse Modeling Framework and exploits Java Emitter Templates for code synthesis. Standard benchmark models show that for generating high-level models underlying state/transition systems significant improvements with respect to CPU time and memory consumption can be realized, ultimately allowing the verification of larger and much more complex systems.

1 Introduction

1.1 Motivation

For excluding fatal damages, hard- and software systems require a formal verification of their (temporal) correctness. Model-based analysis techniques allow to assert the correctness of systems as early as possible in the (re-)design cycle and thereby avoiding costly maldevelopments. Annotated *state/transition systems*

build the foundation for many formal analysis techniques such as (stochastic or probabilistic) model checking for asserting sophisticated functional and quantitative properties or Markovian analysis for obtaining performance and reliability measures. However, for coping with the complexity of todays systems it is more convenient and less error-prone to describe them by means of modularly-structured high-level modelling formalisms, instead of directly specifying annotated state/transition systems. Examples of such formalisms are (extended) Petri nets, UML state charts, process algebraic specifications. Unfortunately a finite state/transition system stemming from some high-level model tends to be exponential in number of states w. r. t. the number of concurrent high-level model activities. This characteristic is commonly known as *state space explosion problem*, it hampers detailed system analyses if not making it impossible in practice. In this context *decision diagrams* which are symbolic representation of finite functions have proven to be very helpful, easing the restriction imposed on the size and complexity of systems to be analyzed. However, in many real-world applications decision diagrams and their related techniques suffer from the so called hill climbing problem: the symbolic representation of state/transition systems itself is naturally very compact but the decision diagram's size increases dramatically during its construction and collapses at the end. As main goal this paper introduces therefore a new technique for generating high-level model's underlying annotated state/transition systems. The obtained symbolic representations may either serve as input for various state-dependent analysis methods or the here proposed algorithms may directly be part of any BDD-based analysis method.

1.2 Contributions and Organization

Contemporary symbolic techniques make use of partial explicit state space exploration and encoding, or implement semantics of high-level model constructs by means of standard BDD-algorithms. However, irrespective the followed strategy one commonly ends up with decision diagrams symbolically representing high-level model's underlying transition functions or parts thereof. For generating a high-level model's reachability set it is these decision diagrams which are employed within symbolic reachability analysis. In Sec. 3 this paper develops new algorithms implementing the semantics of high-level model constructs, but for decision diagrams representing sets of reachable states. Thus contrary to existing techniques these operators can directly be used within partitioned symbolic reachability analysis, rather than making use of symbolically represented transition functions as done before. Overall this strategy may keep the symbolic data structure as flat as possible, most likely to realize run-time and memory advantages over existing techniques.

Besides the functionality of the synthesized algorithms the proposed method has another dimension, which is the development of a modelling environment and the synthesis of algorithms itself, where this paper intends to implement the core of a verification platform for the symbolic analysis of state-based systems. Related issues are discussed in Sec. 4: to be as generic as possible the proposed ideas are implemented on top of the open source software development suite Eclipse and the decision diagram package CUDD [20]. We examplarily implemented a graphical editor for specifying (extended) Petri nets with the Eclipse

Modeling Framework [3] and show how to exploit Java Emitter Templates for synthesizing the algorithms for manipulating decision diagrams. These operators as well as some pre-defined routines are linked with the CUDD-package at compile time, s.t. the obtained executable finally generates a high-level model's reachability graph, a symbolic representation thereof respectively.

For evaluating the performance of the presented approach Sec. 5 analyzes a set of benchmarking models where the obtained results are compared with the runtime data gathered from other competitive symbolic verification tools. Sec. 6 concludes the paper.

1.3 Related work

Over the last decades many different types of decision diagrams have been proposed. However, the obtained structures are mainly extensions of Binary Decision Diagrams (BDDs) [13, 1], and their related algorithms [2]. In this paper we will employ 0-suppressed BDDs [14] extended to the multi-terminal case and enhanced with individual sets of (function-local) input variables. This latter concept which was developed in [12] is of great value for the operators to be developed in this paper, since they are required to operate on multi-rooted decision diagrams [18] as provided by contemporary BDD software packages.

Exploitation of compositionality turns out to be the key to efficiency, when deriving symbolic representations of state/transition systems from high-level models. In practice two composition strategies are known: activity-synchronization [7] and sharing of state variables [11]. For obtaining a high-level model's overall state/transition system one commonly executes a symbolic composition scheme implemented on the basis of standard BDD-algorithms. With these composition schemes which are closely related to the Kronecker-operator-based scheme of [16], one inserts identity structures on those positions which refer to not affected state variables w. r. t. the respective symbolic structure. This has to do with the fact that such variables maintain their values once a (local) transition function is executed. In [10] this idea was taken to the level of decision diagrams by assigning an identity semantics to non-affected variables when computing relational products of states and transition functions. This is another important idea w. r. t. the synthesized algorithms where compositionality is achieved by mapping each high-level activity to its own BDD-operator and non-input variables are treated according to an identity semantics.

As common the proposed procedure makes use of partitioned symbolic reachability analysis (SRA) well known for keeping run-time and space requirements often low [4]. The sequential execution of the synthesized algorithms enables us to employ an *early-update strategy* w. r. t. the set of states to be explored in the next step [11]. The authors of [15] suggest a similar update strategy which they denote as greedy chaining. However, as common for their and all other existing techniques high-level model's transition functions are implemented by means of BDDs employed within SRA, whereas the here proposed technique makes use of perviously synthesized BDD-operators.

2 Preliminaries

2.1 High-level models

The following properties and definitions are assumed to be present: A high-level model M consists of a finite ordered set of discrete state variables (SVs) $\mathfrak{s}_i \in \mathfrak{S}$, where each can take values from a finite subset of the natural numbers. Thus states are given as vectors $\vec{s} \in \mathbb{N}^{|\mathfrak{S}|}$. For convenience we will employ the notation $\vec{s}[i]$ when addressing the value of SV $\mathfrak{s}_i$ in state $\vec{s}$. One may also note that we are dealing with finite models only. Hence, the set of states $\mathbb{S}$ to be represented by a Decision Diagram is a true subset of $\mathbb{N}^{|\mathfrak{S}|}$. Besides SVs a model is assumed to consists of a finite set of activities $\mathcal{A}ct$, where each of these activities $l \in \mathcal{A}ct$ is equipped with its own (activity-local) transition function $\delta_l : \mathbb{S} \to \mathbb{S}$. Activities and SVs are somehow interdependent: on one hand activities manipulate the values of the SVs, whereas on the other hand the specific values of the SVs steer the actual execution of an activity. For exploiting this interdependency we assume that interdependent SVs and activities are connected via the definition of tow connection relations

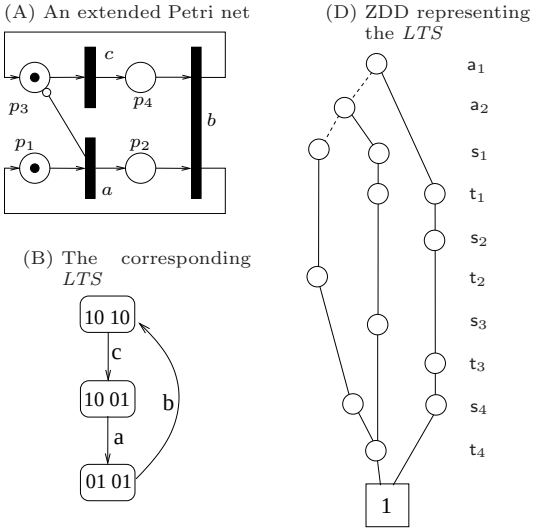
$$\mathcal{C}on \subseteq (\mathfrak{S} \times \mathcal{A}ct) \cup (\mathcal{A}ct \times \mathfrak{S}) \text{ and} \mathcal{T}Con \subseteq (\mathfrak{S} \times \mathcal{A}ct). \tag{1}$$

$\mathcal{C}on$ refers to those pairs, where SVs not only steer an activity's execution, but are actually manipulated by the latter. $\mathcal{T}Con$ refers to those pairs, where SVs only steer the respective activity, i.e. the SVs of these pairs are actually not manipulated once the execution of the activity takes place. Furthermore it is assumed that each element of the above relations is annotated with a weight, extractable through function $\omega_{IO} : \mathcal{C}on \to \mathbb{N}$ and $\omega_T : \mathcal{T}Con \to \mathbb{N}$; for simplicity we use the sloppy notation ω in the following, when referring to one of these functions.

2.2 Low-level models

When executing the transition functions of the high-level model's activities, the latter evolves from one state to another. The implementation of the individual transition functions depends on the model description method, details will follow in Sec. 3.2. For the time being it suffices to assume that starting from the initial state of a high-level model $\vec{s}^{\epsilon}$ the iterative execution of all δ_l-functions in a fixed point computation yields a transition relation $T \subseteq (\mathbb{S} \times \mathcal{A}ct \times \mathbb{S})$, where this step is commonly denoted as state space exploration. The obtained labelled transition system (*LTS*) T, which we already informally addressed as state/transition system (state/transition system) defines the semantic model of any (finite) high-level modelling technique as considered in this paper. One may note that most sophisticated high-level modelling techniques have Turing-power, thus finiteness of the underlying transition systems is not decidable, hence termination of state space exploration unknown.

A very simple example of a high-level model and its underlying *LTS* is shown in Fig. 1 A and B: Part (A) shows a Petri net (PN) extended with an inhibitor arc, the latter connects place p_3 with activity a (the circle-headed edge). This PN consists of the activity set $\mathcal{A}ct := \{a, b, c\}$ and the set of SVs $\mathfrak{S} := \{p_1, p_2, p_3, p_4\}$ and the connection relation $\mathcal{C}on$ and $\mathcal{T}Con$, those elements are given by the connecting edges, $\mathcal{T}Con$ solely contains one element, namely the pair (p_3, a). Thus



l	$\vec{a}$	$\vec{s}$	$\vec{t}$	f_M
	a_1a_2	$s_1s_2s_3s_4$	$t_1t_2t_3t_4$	
a	00	1001	0101	1
c	01	1010	1001	
b	10	0101	1010	

(C) Binary encodings of the *LTS*

Figure 1: From a PN to the ZDD-based representation of its *LTS*

each state can be represented by a 4-dimensional vector, where each element i refers to the number of tokens contained in the respective place p_i. For the order $p_1 \prec p_2 \prec p_3 \prec p_4$ the initial state $\vec{s}^{\epsilon}$ is given by $(1, 0, 1, 0)$. Each of the PN's activities make use of its own transition function, which can be executed once the activity's input places contain a sufficient number of tokens and the inhibitor arcs are not in effect (see Sec. 3.2 for an elaborated discussion on this). For the initial state only the execution of activity c is possible, yielding the transitions $(1, 0, 1, 0) \xrightarrow{c} (1, 0, 0, 1)$. The *LTS* obtained from state space generation is depicted in Fig. 1.B.

2.3 The ZDD data structure

Reduced ordered BDDs [2] are directed acyclic graphs for canonically representing Boolean functions: $\mathbb{B}^{|\mathcal{S}|} \to \mathbb{B}$, where $\mathcal{S} := \{s_1, \ldots, s_n\}$ is a finite set of (Boolean) input or function variables and $\mathbb{B}$ is the Boolean set $\{0, 1\}$. To be generic as possible we consider n-ary pseudo-Boolean functions, i.e. functions of the type $f : \mathbb{B}^n \to \mathbb{F}$, where $\mathbb{F}$ is a finite set, e.g. $\mathbb{F} \subset \mathbb{R}_{\geq 0}$. For canonically representing such functions we exploit reduced ordered 0-suppressed Multi-terminal Binary Decision Diagrams (ZDDs) [12] which are *zero-suppressed* BDDs [14] extended to the *multi-terminal* case. In a zero-suppressed BDD the skipping of a variable –each inner node is associated with a variable– means that this variable takes the value 0. The Shannon expansion for Boolean functions gives that a ZDD's graph and a set of Boolean input variables only together canonically represent a Boolean function. In the following we will use therefore the notation Z<$\mathcal{S}$> when emphasizing the set of input variables $\mathcal{S}$ decisive for ZDD Z. However, within shared BDD-environments as provided by packages such as CUDD

[20], ZDD-nodes loose their uniqueness as soon as the ZDDs are defined on different sets of function variables. To solve this problem [12] introduced the concept of partially shared ZDDs and algorithms for manipulating them. The idea is as follows: When working with partially shared ZDDs, i.e. with ZDDs having different sets of input variables, one also iterates over the input variables of the operand ZDDs. This allows one to assign a specific semantics to each visited but skipped variable on the current path, namely either Bryant's standard *don't care*- or Minato's 0-suppression variable reduction rule; a don't-care node is a node whose `then`- and `else`-children are identical. From reduced ordered BDDs such nodes are removed, since the node-labeling variable is obviously irrelevant for the function value, the respective variable $\mathtt{s}_i$ could either be interpreted as true or false without changing the function value. With 0-suppressed variables the skipping implies that the `then`-child of the respective node is the terminal *0-node*, whereas the `else`-child is given by the currently visited node. This will be considered for the operators to be synthesized, but instead of following a *dnc*-semantics a identity-semantics applies once the respective operator hits a non-input variable.

2.4 ZDD-based representation of LTS

Exhaustive state space exploration yields a successor-state relation $T \subseteq (\mathbb{S} \times \mathcal{A}ct \times \mathbb{S})$. Each transition contained in T can now be encoded by applying a binary encoding function $\mathcal{E}$. The individual bit positions correspond to the input variables of the ZDD-environment, where the $n_{\mathcal{A}ct}$ variables of $\vec{\mathtt{a}}$ hold the binary encoded activity label, and the $2n$ $\mathtt{s}$ and $\mathtt{t}$-variables the encodings of the source and target states. As common the variables are ordered in an interleaved fashion [7]:

$$\mathtt{a}_1 \prec \ldots \prec \mathtt{a}_{n_{\mathcal{A}ct}} \prec \mathtt{s}_1 \prec \mathtt{t}_1 \prec \ldots \prec \mathtt{s}_n \prec \mathtt{t}_n. \tag{2}$$

In total this encoding yields a function table constructible for each finite *LTS*, where a function of this kind can canonically be represented by a ZDD $\mathsf{Z}<\vec{\mathtt{a}}, \vec{\mathtt{s}}, \vec{\mathtt{t}}>$.

For exemplification please refer again to Fig. 1: Subfigure (B) illustrates the *LTS* as obtained by carrying out state space exploration in case of the PN of sub-figure (A). Table C shows the (binary) encodings of the individual transitions, where this table constitutes the function table of a Boolean function. The ZDD constructible from this function table is shown in sub-figure D, where a dashed (solid) arrow indicates the value assignment 0 (1) to the corresponding Boolean variable on the respective path. For simplicity we omitted the terminal *0-node*, including all of its in-going edges.

2.5 Notation

In the following mathematical calligraphy is related to the syntactical objects of high-level models and variables of decision diagrams and their sets are referred to by non-serif fonts. Integer values, state vectors etc. will be addressed by standard type setting, e.g. $\mathfrak{s}_i$ refers to SV i, $\mathtt{s}_i$ refers to a boolean variable of a decision diagram and $\vec{s}[i]$ denotes the integer at position i as contained in state $\vec{s}$.

Algorithm 1 Activity-wise organized SRA

$SRA(\vec{s}, \mathcal{A}ct)$
(0) Z_R<$\mathsf{s}_1, \ldots \mathsf{s}_n$>$:= \emptyset, \mathsf{Z}_{tmp}$<$\mathsf{s}_1, \ldots \mathsf{s}_n$>$:= \emptyset$
(1) Z'_R<$\mathsf{s}_1, \ldots \mathsf{s}_n$>$:= \mathtt{MakeZDD}(\vec{\mathsf{s}}, \mathcal{E}(\vec{s}^{\,\epsilon}))$
(2) DO
(3) $\quad \mathsf{Z}_R := \mathsf{Z}'_R \vee \mathsf{Z}_R$
(4) $\quad$ FOR $l \in \mathcal{A}ct$ DO
(5) $\quad\quad \mathsf{Z}_{tmp} := \mathtt{Img}_l(\mathsf{Z}'_R, \mathcal{S})$
(6) $\quad\quad \mathsf{Z}'_R := \mathsf{Z}_{tmp} \vee \mathsf{Z}'_R$
(7) $\quad$ END
(8) $\quad \mathsf{Z}'_R := \mathsf{Z}'_R \setminus \mathsf{Z}_R$
(9) END UNTIL $\mathsf{Z}'_R = \emptyset$
(10) RETURN Z_R

3 Functionality of Synthesized Operators

For obtaining a high-level model's reachability set, its ZDD-based representation respectively, the proposed approach executes a partitioned symbolic reachability analysis (SRA) [4], which we organize in an activity-wise manner (cf. Algo. 1). However, in contrast to existing techniques the proposed approach is based on the execution of a BDD-operator $\mathtt{Img}_k$. Informally speaking this operator computes for a given set of source states ($\mathbb{D} \subseteq \mathbb{S}$) a set of successor states ($\mathbb{T} \subseteq \mathbb{S}$) w. r. t. high-level activity k. Since the set $\mathbb{T}$ is commonly denoted as image of $\mathbb{D}$ w. r. t. k we will denote operator $\mathtt{Img}_k$ as the (one-step) image operator of k. The iterative execution of all of a high-level model's operators in a fixed point computation delivers a system's reachability set; given that this set is finite which is in general not decidable for most high-level modelling techniques. The obtained symbolic representation which represents the high-level models reachability set is than employed for efficiently generating a model's underlying *LTS* which will be done once again on basis of synthesized ZDD-algorithms. In the following we will give details about the functionality of the employed algorithms and ZDD-operators to be synthesized.

3.1 Symbolic Reachability Analysis (SRA)

The algorithm implementing partitioned SRA is given in Listing 1. it makes use of the following three ZDDs

- Z'_R represents the set of states reached in the last round of image computations
- Z_R represents the set of states reached so far and
- Z_{tmp} represents the one-step image w. r. t. the activity currently under consideration and w. r. t. the states represented by Z'_R.

These structures take the n variables $\{\mathsf{s}_1, \ldots \mathsf{s}_n\}$ as input variables, for encoding each states ($n := \sum_{i:=1}^{|\mathfrak{S}|} \lceil \log_2(\max_{\mathbb{S}}(\vec{s}[i])) \rceil$.) The functionality of the algorithm

is as follows: In line 1 the sets of reachable states are initialized where Z'_R takes the encoding of the high-level model's initial state $\vec{s}^{\,\epsilon}$. Starting from this state exploration is executed in an activity-oriented fashion by executing the ZDD-operators one by another and thereby repetitively computing the-one-step reachability set w. r. t. the current activity (line 4-7). This is done until the global fixed point $\mathsf{Z}'_R \setminus \mathsf{Z}_R = \emptyset$ is reached (line 2-9).

3.2 One-step reachability set computation

As pointed out before with the proposed approach each of the high-level model's activities is mapped to its own specific ZDD-operator implementing the computation of the one-step image w. r. t. this very activity and a set of states. For keeping the discussion as simple as possible we will follow the semantics of (extended) k-bounded Petri nets, where one may recall that this bound k is not known a priori to state space construction. In such a setting the algorithm for computing the one-step reachability set w. r. t. an activity l has to implement subtraction and addition of tokens but on the level of ZDDs and their input variables. Furthermore the ZDD-operator is required to incorporate the functionality of inhibitor arcs, s.t. for dedicated values of SVs an activity's execution is suppressed. In case of non-affected SVs the operator must in-cooperate an identity semantics, since the related positions do not change their value when the respective activity is executed. To formalize this setting the following definitions are required:

3.2.1 Definitions

Each $\mathfrak{s}_i \in \mathfrak{S}$ will be represented by its own tuple $< \mathsf{s}_1^i, \ldots \mathsf{s}_{n_i}^i >$ of globally visible boolean variables. In the following we will make use of the notation $\vec{\mathsf{s}}^{\,i}$ when referring to the respective tuple. It is straight forward to order the tuples according to their indices which yields the total order: $\mathsf{s}_1^1 \prec \mathsf{s}_2^1 \prec \ldots \prec \mathsf{s}_{n_i}^1 \prec \ldots \prec \mathsf{s}_1^{|\mathfrak{S}|} \prec \ldots \prec \mathsf{s}_{n_{|\mathfrak{S}|}}^{|\mathfrak{S}|}$ for a ZDD-environment. One may note that the number of variables required for encoding SV $\mathfrak{s}_i$ is not known a priori to state space generation. This requires a dynamic strategy where one allocates a new most significant bit as soon as possible, namely once an overflow for $\mathfrak{s}_i$ occurs. To do so one simply needs to insert a new Boolean variable representing the new most significant bit w. r. t. SV $\mathfrak{s}_i$. Contrary to standard BDDs, ZDDs have here the nice feature that for such cases the structure of the ZDD-graph remains untouched, the newly allocated variable is automatically 0-assigned for all previously encoded states.

The relations defined Eq. 1 allow to define the following sets w. r. t. an activity l: $\mathfrak{S}_l^{in}$ referring to l's set of input variables, $\mathfrak{S}_l^{out}$ referring to its set of output variables, $\mathfrak{S}_l^{test}$ referring to its set of test variables and $\mathfrak{S}_l^I$ as its set of independent variables. As illustrated in Sec. 2.4 each SV is encoded by a set of boolean variables, thus the set of activity-local variables as defined above can be extended to the boolean setting:

$$\begin{aligned} &\bullet\, \mathsf{D}_l^{in} := \{\vec{\mathsf{s}}^{\,i} | \mathfrak{s}_i \in \mathfrak{S}_l^{in}\} \qquad \bullet\, \mathsf{I}_l := \{\vec{\mathsf{s}}^{\,i} | \mathfrak{s}_i \in \mathfrak{S}_l^{I}\} \\ &\bullet\, \mathsf{D}_l^{out} := \{\vec{\mathsf{s}}^{\,i} | \mathfrak{s}_i \in \mathfrak{S}_l^{out}\} \quad \bullet\, \mathsf{T}_l := \{\vec{\mathsf{s}}^{\,i} | \mathfrak{s}_i \in \mathfrak{S}_l^{test}\}. \end{aligned} \tag{3}$$

For exemplification please refer to Fig. 1.A: according to the above discussion activity a takes places p_1 on its sets of dependent input SVs ($\mathfrak{S}_a^{in} = \{p_1\}$) and the place p_2 on its sets of dependent output SVs ($\mathfrak{S}_a^{out} = \{p_2\}$), whereas the places p_3 and p_4 appear on a's set of independent SVs ($\mathfrak{S}_a^{I} = \{p_3, p_4\}$) and p_3 appearing also on its set of test variables $\mathfrak{S}_a^{test}$. Since each SV p_i is encoded by a single bit only the sets of boolean variables are defined accordingly: $\mathsf{D}_a^{in} := \{\mathsf{s}_1\}$, $\mathsf{D}_a^{out} := \{\mathsf{s}_2\}$, $\mathsf{I}_a := \{\mathsf{s}_3, \ldots, \mathsf{s}_5\}$ and $\mathsf{T}_a := \{\mathsf{s}_3\}$.

3.2.2 Model world

In the following we will make the discussion much more concrete w. r. t. a model's execution semantics to be implemented by the synthesized operators.

The high-level model's evolution from state to state is achieved by executing the individual activities, where an activity l is executable *iff* its predicate function $pred_l$ evaluates to true. For the model world considered in this paper we have:

$$pred_l(\vec{s}) := true \Leftrightarrow \forall \mathfrak{s}_i \in \mathfrak{S}_l^{in} \cup \mathfrak{S}_l^{test} : \omega(l, \mathfrak{s}_i) \leq \vec{s}[i], \tag{4}$$

where ω is the weight-returning function. In case $pred_l(\vec{s}) = true$ one says that activity l is enabled in state $\vec{s}$, referred to by the notation $\vec{s}\,[> l$. If $pred_l(\vec{s}) = false$ one writes $\vec{s}\,[\not> l$. By executing enabled activities one successively generate the model's underlying transition relation. The activity-specific transition function $\delta_l : \{\vec{s} \in \mathbb{S} \mid \vec{s}\,[> l\} \rightarrow \mathbb{S}$ is defined as follows: $\delta_l(\vec{s}) := \vec{t}$ where

$$\vec{t}[i] := \begin{cases} \vec{s}[i] - \omega(\mathfrak{s}_i, l) & \Leftrightarrow \mathfrak{s}_i \in \mathfrak{S}_l^{in} \cap \neg\mathfrak{S}_l^{out} \\ \vec{s}[i] + \omega(l, \mathfrak{s}_i) & \Leftrightarrow \mathfrak{s}_i \in \mathfrak{S}_l^{out} \cap \neg\mathfrak{S}_l^{in} \\ \vec{s}[i] - \omega(\mathfrak{s}_i, l) + \omega(l, \mathfrak{s}_i) & \Leftrightarrow \mathfrak{s}_i \in \mathfrak{S}_l^{in} \cap \mathfrak{S}_l^{out} \\ \vec{s}[i] & else \end{cases} \tag{5}$$

Together with the initial state $\vec{s}^{\,\epsilon}$ assigning an initial value to each SVs the set of all activity-specific predicate and transition functions allows one to construct the set of reachable states $\mathbb{S}$ which we define inductively as follows:

$$\begin{array}{l} \vec{s}^{\,\epsilon} \in \mathbb{S} \\ (\vec{s} \in \mathbb{S} \wedge \exists l \in \mathcal{A}ct : pred_l(\vec{s}) = true) \Rightarrow \delta_l(\vec{s}) \in \mathbb{S} \end{array} \tag{6}$$

By employing the activity-specific transition function δ_l in a loop the former can be extended to the case of sets $\Delta_l : 2^{|\mathbb{S}|} \longrightarrow 2^{|\mathbb{S}|}$ and which we we define as follows:

$$\Delta_l(\mathbb{D}) := \begin{cases} \emptyset & \Leftrightarrow \nexists \vec{s} \in \mathbb{D} : \vec{s}\,[> l \\ \mathbb{T} \subseteq \mathbb{S} & else \end{cases} \tag{7}$$

where $\mathbb{D}$ is some set of states. It is exactly the functionality of Δ_k which will be implemented by operator $\texttt{Img}_k$ but on the level of ZDDs.

3.2.3 ZDD-operator for one-step reachability set computation

Following standard BDD-algorithms [2] `Img` recurses on the nodes of the as argument passed ZDD Z_R in a depth-first-search manner. ZDD Z_R represents some set of states, namely all states reached so far. Contrary to other BDD-algorithms operator `Img` needs to recurse on all dependent input/output and test variables. This has to do with the 0-suppression of variables as well as with the bit-wise

addition, subtraction or inhibition of activity executions. Hence, the synthesized operator $\texttt{Img}_k$ stops at least for each ZDD variable $\mathsf{s}_j^i \in \mathsf{D}_k^{in} \cup \mathsf{D}_k^{out} \cup \mathsf{T}_k$. The Boolean value associated with the current variable s_j^i and w. r. t. the currently traversed path is only known once the recursion has descended to the child-nodes, where $\mathsf{s}_j^i = 1$ in case the algorithm recurses to the `then`-child and $\mathsf{s}_j^i = 0$ otherwise. Since we are dealing with bit-wise addition and subtraction the operator must consider carries to be shifted to the next variable. We therefore make use of a least-significant-bit-first order defined on the variables $\mathsf{s}_j^i \in \vec{\mathsf{s}}^i$, where the s-tuples are ordered according to their indices i.

Instead of discussion a concrete implementation of operator $\texttt{Img}_k$ we briefly indicate the recursive behavior w. r. t. the currently visited variable s_j^i, we assume that the reader is familiar with Bryant's standard BDD-algorithms [2].

(a) Test variables ($\mathsf{s}_j^i \in \mathsf{I}_{\mathsf{l}}^{\mathsf{test}}$)
In order to implement Eq. 4 one needs to test for integer values associated with SV $\mathfrak{s}_i$. For tracking integer values the operator `Img` makes use of a local variable *tmpSum* which book-keeps the value currently associated with SV $\mathfrak{s}_i$. For $tmpSum + 2^{j-1} \cdot \mathsf{s}_j^i \geq K$ it is straight-forward to terminate the recursion and returning the terminal *0-node*, where K is some pre-defined value as returned by the weight-function ω. In case $sum + 2^{k-1} \cdot \mathsf{s}_j^i < K$ the value held by $\mathfrak{s}_i$ does not interfere with the execution of activity l, s.t. one may recurse further. However, as pointed out above the actual value of variable s_j^i is only known once we recurse deeper. Hence the actual value of *tmpSum* is checked in the next recursion.

(b) Input variables ($\mathsf{s}_j^i \in \mathsf{D}_l^{in}$)
Analogously to the above setting one must check if the value currently associated with a SV $\mathfrak{s}_i$ is larger than some bound K as given by the weight-function ω, if this is the case one may descend to the next variable otherwise the recursion can be terminated by returning the terminal *0-node* .

(c) Output variables ($\mathsf{s}_j^i \in \mathsf{D}_l^{out}$)
When returning from the recursion it must be decided whether a new node must be allocated for the current variable or not. To do so one must know if a carry was handed over from the previous recursion and whether the addition makes the current variable 0 assigned or not. Also it might be necessary for merging the results as obtained once returning from the `then`- or `else`-recursion.

(d) Independent variables ($\mathsf{s}_j^i \in \mathsf{I}_l$)
This case is quite special, since variable s_j^i is in principle not a function variable for the transition function defined by activity l. Due to Eq. 5 such positions can be handled according to an identity semantics. This is straight-forward, one only allocates a node for variable s_j^i if there is already a node on the current path. Finally one may note that variables may appear on different sets. Hence a mixture of the above basic rules might be applicable.

3.2.4 ZDD-operator for *LTS* generation

Once the above explained operators have helped to generate a high-level model's reachability set, one may generate the overall models transition relation, a ZDD-based representation thereof respectively by making use of the respective ZDD-

operator MakeTrans. Each transition of a high-level model with source state $\vec{s}$, activity l and target state $\vec{t} := \delta_l(\vec{s})$ is a triple $(\vec{s}, l, \vec{t})$. The set of all such triples yields an *LTS* $T \subseteq \mathbb{S} \times \mathcal{A}ct \times \mathbb{S}$. As illustrated in Sec. 2.4 each of such triples to be generated by an instance of $\texttt{MakeTrans}_l$ will be encoded as a path in a ZDD $\mathsf{Z}_l < \vec{\mathsf{a}}, \vec{\mathsf{s}}, \vec{\mathsf{t}} >$, s.t. the union over all Z_l represents a high-level model's complete transition relation. At first this requires to introduce the missing a and t-variables into the ZDD-environment, where the order as defined in Eq. 2 applies. Recursing on the previously generated ZDD Z_R and allocation of nodes if necessary enables $\texttt{MakeTrans}_l$ to generate a symbolic representation of all transitions as induced by high-level activity l. It is easy to see, that the implementation of $\texttt{MakeTrans}_l$ is straight-forward one simply needs to follow the ideas of $\texttt{Img}_l$, but extended to the case of a and t-variables.

3.3 Remarks on correctness and completeness

3.3.1 Reachability set construction

Given a ZDD Z_R representing a set of states $\mathbb{S}'$ the operator $\texttt{Img}_l$ terminates its recursion by returning the terminal *0-node* as soon as $pred_l(\vec{s}) = false$ holds. For all other paths it constructs the encodings of the respective successor state $\delta_l(\vec{s})$. Thus the execution of the inner FOR-loop of Algo. 1 (line 4-7) constructs the set

$$\mathbb{S}'_l = \{\vec{t} := \delta_l(\vec{s}) | \vec{s} \in \mathbb{S}' \wedge pred_l(\vec{s}) = true\}$$

One iteration of the outer DO-UNTIL loop of Algo. 1 (line 2 - 9) delivers all states reachable from $\mathbb{S}'$ in a single step, where line (6) successively computes the union over all newly reached $\mathbb{S}'_l$ ultimately collected in structure Z_R. This is repeated until no new state can be found, i.e.

$$\nexists \vec{s} \in \mathbb{S} : pred_l(\vec{s}) = true \wedge \delta_l(\vec{s}) \notin \mathbb{S}\}$$

holds which is exactly what was defined in Eq. 6 for the reachability set to be generated.

The above procedure only terminates *iff* $\mathbb{S}$ is finite. Most high-level modelling formalism possess Turing-power, hence finiteness of $\mathbb{S}$ is not decidable and thus termination of reachability set construction is not guaranteed, neither for the explicit nor for the symbolic setting.

3.3.2 LTS-generation

Once reachability set construction terminates the set of all reachable $\mathbb{S}$ has been constructed. For each high-level activity l one executes now an operator $\texttt{MakeTrans}_l$, which constructs a ZDD Z_T^l for representing all tuples $(\vec{s}, l, \vec{t})$ where $pred_l(\vec{s}) = true$ holds. For paths with $pred_l(\vec{s}) = false$ one may once again terminate the recursion by returning the terminal *0-node*. It is easy to see that the union over all such paths and over all Z_T^l delivers the high-level models overall transition relation, its ZDD-based representation respectively.

4 Implementation

In the following we give brief details about the model editor and the synthesis of ZDD-operators.

4.1 Yet another Model Editor

It is useful to allow third parties to easily extended the tool bench with other high-level modelling formalisms or to incorporate their BDD-based analysis methods. As the BDD-operators for symbolic image computations are directly derived from a high-level model, any platform must not only offer rapid development of graphical editors, but also has to allow a tight integration to any C/C++-environment as needed by contemporary BDD-package such as CUDD. Eclipse with its C++-extension CDT offers such a platform enabling the intuitive creation of graphical editors with the graphical modelling framework (GMF). At first step one specifies a class diagram, which consists of the high-level modelling formalism's syntactical objects, here places, activities and their directed connecting edges. These entities are finally mapped to previously defined graphical representations, enabling an automatic generation of a graphical user front end, where the GMF auto-generates the Java-code for the graphical editor. As main feature it is important to note that each high-level model is mapped to some Java-code according to the predefined rules. This customized code ultimately allows us to access the different syntactical objects of a high-level model and retrieve the information as used in the process of operator synthesis.

4.2 Synthesizing ZDD-operators

As pointed out above access to a user-defined model is achieved via the EMF auto-generated Java-routines. For generating the C++-based BDD-operators additional Java-code is supplied by us, where we made use of Java Emitter Templates (JET). JET allow the specification of a Java skeleton, as well as the C++ output written to a source-code file. For exemplification one may refer to Algo. 2. The specified JET consists of a Java-skeleton (upper-case type setting) which controls the output of C++-code (lower-case type setting). For keeping the discussion as simple as possible we solely concentrated on the generation of code which implements the functionality of the $pred_k$-function (Eq. 4) but on the level of ZDDs.

Since a ZDD-operator is synthesized for each high-level activity, Algo. 2 cycles through all high-level activities (line 1-25). In line 2 we specify the function header of the ZDD-operator to be written into the respective C++ source-code file. As next starting from line 3 the algorithm iterates through all places, SV respectively which are connected with the activity currently under consideration. The required information, such as the index of the SV, its type (input, output or test), and the weight of the connecting edges can be obtained via the EMF auto-generated Java-routines. As pointed out in Sec. 2.4 each SV is encoded by a ordered set of variables $\vec{\mathsf{s}}^i$. Each of these variables s^i_j need now to be processed which is done in the loop of line 7-25. Depending on the type of this boolean variable, i.e. if it is an input, output or test variable w. r. t. the current activity the code to be generated changes (line 12-16, line 17-21 and line 22-24).

Algorithm 2 Generation of firing conditions

```
(0)  ......
(1) FOREACH ACTIVITY T {
(2)  Node Img_T(Node node, var v_c, int tmpSum, int carry){
(3)  ITERATE THROUGH PLACES_CONNECTED_WITH_T {
(4)    GET PLACEDESCIPTOR = {SV_INDEX, CONN_TYPE, CONN_WEIGHT};
(5)    CURRENTARCPLACELIST = ALL DESCRIPTORS OF CURRENT PLACE;
(6)    INDEX = 0;
(7)    FOREACH POS IN THIS PLACES BITS{
(8)      if(v_c == POS){
(9)        isDependentSV = true;
(10)       if(var(node) == v_c)currLevelToken = (1 << INDEX);
(11)       ITERATE THROUGH CURR_PLACE_ARC_LIST {
(12)         IF(ARCTYPE = INPUT){
(13)           if(tmpSum < ARCWEIGHT)return(0-node);
(14)           weightSet = (ARCWEIGHT&(1 << INDEX))! = 0;
(15)           op = bitwiseSubstract;
(16)         }
(17)         IF(ARCTYPE = OUTPUT){
(18)           if(tmpSum > CAPACITY)HandleNewMSB(v_c);
(19)           weightSet = (ARCWEIGHT&(1 << INDEX))! = 0;
(20)           op = bitwiseAdd;
(21)         }
(22)         ELSE IF(ARCTYPE = INHIBITOR){
(23)           if(tmpSum >= ARCWEIGHT)return(0-node);
(24)         }
(25) ......
```

It is also important to note that a boolean variable $\mathtt{s}_j^i$ can be of different types, hence one needs to iterate overall different pairs of connections (line 11 -25). To keep the remaining part of the operator identical for all variables and variable types the synthesized operator makes use of some local variables which are set as specified in Algo. 2. In particular we employ:

- `op` holds which binary operation is to be used upon computation of the current result bit. Obviously this is a binary subtraction for the case of input variables and binary addition for output variables.
- `dependentSV` determines whether the current variable is part of any SV affected by this activity or not. If the variable is independent its bit value won't change on application of this activity.
- `currentLevelToken` holds the token value resulting when the current bit would be set. This is solely affected by current variables relative offset j w. r. t. tuple $\vec{\mathtt{s}}^i$. This information is useful since the need to update *tmpSum* as pointed out before accordingly.
- `tmpSum` is the aforementioned variable for tracking the values of the SVs and terminate recursion if necessary.

The JET allows us to generate the C++ code required for implementing each of the `Img`-operators. This code is combined with pre-defined routines and headers, e.g. for executing the SRA, as well as with the employed BDD-package CUDD, yielding an executable whose execution carries out symbolic state space generation for the respective high-level model.

5 Empirically evaluating the new scheme

5.1 Preliminary Remarks

For empirically evaluating the proposed technique models from the area of stochastic verification have been taken. With stochastic verification techniques one commonly specifies a system by means of stochastic or Markovian extensions of well known modelling techniques, e.g. Stochastic Petri Nets (SPN), Stochastic Process Algebra (SPA), among many other. There high-level model's underlying stochastic *LTS* can directly be interpreted as Discrete or Continuous Time Markov Chain, whose solution allows to compute system measures of interest or to quantify a formulae to be checked on a system. To adapt to the need of a SPN the so far presented framework must solely be alter by probabilities or rates attached to the activities of the high-level model. On the level of symbolic representations this is straight-forward: the individual transition probabilities or rates are interpreted as function values and stored in the terminal nodes of the symbolic structures. As competitors we choose the MTBDD-based stochastic model checker Prism [17] and the ZDD-based engine of the Moebius Modelling Framework [11]. This choices is based on the fact that these tools also employ the BDD-library CUDD [20] and make use of a SRA. At first we briefly sketch the symbolic techniques employed in the competitors.

(a) Semi-symbolic scheme employed in Moebius:
The ZDD-engine of the Moebius modelling framework carries out a semi-symbolic generation scheme. Semi-symbolic since the method requires explicit exploration and encoding of a high-level model's underlying *LTS*. However, as common for semi-symbolic techniques it exploits the compositional structure of high-level models which makes the explicit handling of transitions partial. Overall the efficiency of the semi-symbolic technique depends on a model's structure, where for less efficient cases one spots explicit state space generation and encoding and for the nicely structured models pure DD-based manipulations as main source for run-time and peak memory consumption.

(b) Fully-symbolic technique as employed in Prism:
Like the framework proposed in this paper, the stochastic model checker Prism makes use of a fully symbolic state graph generation technique. Such approaches require operators of their modelling methods to possess BDD-based semantics in order to construct a model's underlying *LTS*. Prism makes use of a s state-based language which follows the Reactive Modules formalism of Alur and Henzinger. Symbolic semantics for constructing the modul-local *LTS*, their MTBDD-based representations respectively is realized on the basis of standard BDD-algorithms and explicitly bounded sizes of model variables. For obtaining the overall models transition relation activity-synchronization as discussed in Sec. 1.3 applies. Hence one may spot symbolic reachability analysis as the main source of run-time and memory consumption, where Prism makes use of a standard breath-

			ZDD-based Moebius		MTBDD-based Prism		ZDD-based new scheme	
N	$\#states$	$\#trans.$	t_g (sec.)	mem_{pk}	t_g (sec.)	mem_{pk}	t_g (sec.)	mem_{pk}
Polling								
10	1.54E+004	8.96E+004	0.02	15,682	0.06	3,922	0.007	6,132
15	7.37E+005	1.58E+007	0.06	64,605	0.16	12,796	0.013	13,286
30	4.83E+010	7.65E+011	?	?	1.12	66,558	0.055	56,210
Kanban								
15	4.70E+010	6.13E+011	47.29	26,039,756	83.72	859,203	0.656	446,614
20	8.05E+011	1.10E+013	?	?	536.12	1,997,252	1.733	974,988
25	7.68E+012	1.08E+014	?	?	1532.75	3,866,941	9.338	1,808,940
Tandem								
1024	2.10E+006	7.33E+006	225.66	58,176,776	18.43	68,005	2.263	283,094
2048	8.38E+006	2.93E+007	?	?	117.66	158,334	15.746	623,420

Table 1: Run-time and peak memory consumption

first-search scheme.

5.2 Results

For carrying out the benchmarking well known performance models have been chosen: the Cyclic Sever system (Polling) [9], the Kanban manufacturing model (Kanban) [5] and the Tandem Queueing model (Tandem) [8]. The experiments were executed on a Intel Duo Core platform (2 GHz), equipped with 1 GByte of RAM and a Linux OS. Tab. 1.A shows the model's scaling parameter N, the number of states and transition the different state/transition system consists, the overall state graph construction times (t_g) and the peak memory requirements in BDD-nodes (16 Bytes each). As one can see the activity-local scheme is sometimes not capable of constructing a high-level model's underlying *LTS* which has to do with explicit generation and encoding of transitions. In cases where this is done for only few transitions the scheme delivers acceptable results, but nevertheless the proposed scheme clearly outperform this semi-symbolic method. Run-time overheads of the Prism tool are imposed first of all by the non-partitioned SRA scheme and secondly by the symbolic composition scheme, where in both cases standard BDD-algorithms are employed. However, it is interesting to note, that for the Polling and Tandem model Prism has much lower peak memory requirements. This clearly has to do with the dense state enumeration of the respective models, since from other experiments it is known that the usage of ZDD most of the times leads to a speed-up and reduction in memory-requirements by a linear factor. However, the here proposed technique can easily be extended to the case of MTBDDs allowing further investigations.

6 Conclusion

This paper presents an approach for generating high-level model's underlying state/transition systems. In contrast to existing symbolic techniques it is not a set of decision diagrams, but a set of synthesized operators implementing a high-level model's transition functions and employed for executing partitioned symbolic reachability analysis. As demonstrated, the presented scheme outperforms other contemporary stochastic model checkers Overall this paper yields the core for a new competitive verification platform for the symbolic analysis of state-based systems, which we implemented on top of the Eclipse Modelling Framework and the CUDD-package.
In future work we like to incorporate the proposed ideas directly into BDD-based model checking routines for gaining further run-time and memory advantages ultimately allowing a comparison to model checkers such as SMV [19] or SAT-based [6] methods.

Acknowledgement
I thank Juri Baumberg who implemented the presented ideas.

References

[1] S.B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978.

[2] R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[3] Frank Budinsky, Ed Merks, and David Steinberg. *Eclipse Modeling Framework 2.0 (2nd Edition)*. Addison-Wesley Professional, 2008.

[4] 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.

[5] 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.

[6] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[7] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for symbolic model checking in CCS. *Distributed Computing*, 6(3):155–164, 1993.

[8] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi Terminal Binary Decision Diagrams to Represent and Analyse Continuous Time Markov Chains. In *Proc. of NSMC'99*, pages 188–207. Prensas Universitarias de Zaragoza, 1999.

[9] O. Ibe and K. Trivedi. Stochastic Petri net models of polling systems. *IEEE Journal on Selected Areas in Communications*, 8(9):1649–1657, 1990.

[10] K. Lampka. A new algorithm for partitioned symbolic reachability analysis. *ENTCS 223*, Workshop on Reachability Problems, 2008.

[11] K. Lampka and M. Siegle. Activity-Local State Graph Generation for High-Level Stochastic Models. In *MMB'06*, pages 245–264, 2006.

[12] K. Lampka, M. Siegle, J. Ossowskis, and C. Baier. Partially-shared zero-suppressed Multi-Terminal BDDs: Concept, Algorithms and Applications, 2008. Accepted for publication in the journal FMSD, cf. ftp.tik.ee.ethz.ch/pub/publications/TIK-Report-289.pdf.

[13] C.Y. Lee. Representation of Switching Circuits by Binary-Decision Programs. *Bell Systems Technical Journal*, 38:985–999, 1959.

[14] S. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *Proc. of the 30th Design Automation Conference (DAC)*, pages 272–277, Dallas (Texas), USA, 1993. ACM / IEEE.

[15] Enric Pastor, Jordi Cortadella, and Oriol Roig. Symbolic analysis of bounded petri nets. *IEEE Trans. Comput.*, 50(5):432–448, 2001.

[16] Brigitte Plateau. On the stochastic structure of parallelism and synchronization models for distributed algorithms. In *Proc. of SIGMETRICS '85:*, pages 147–154, New York, NY, USA, 1985. ACM Press.

[17] PRISM web page. www.prismmodelchecker.org.

[18] T. Sasao and M. Fujita, editors. *Representations of Discrete Functions.* Kluwer Academic Publishers, 1996.

[19] The Cadence SMV Model Checker web page. www.kenmcmil.com/smv.html.

[20] F. Somenzi. CUDD: Colorado University Decision Diagram Package, Release 2.3.0. User's Manual and Programmer's Manual, 1998.