Re-visiting Partitioned Symbolic State Graph Generation for high-level models

Kai Lampka

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

Re-vised: June 8, 2010

TIK-Report 308,
ftp://ftp.tik.ee.ethz.ch/pub/publications/TIK-Report-308.pdf

Abstract Modeling formalisms allow one to compactly describe complex systems. But, once it comes to the actual analysis the high-level descriptions are commonly required to be transformed into flat state-transition systems (ST-systems), e.g. for asserting system properties by means of model checking techniques. Decision Diagrams and their algorithms are well accepted for carrying out the above mentioned transformation. This paper introduces a combined procedure for efficiently constructing a Binary Decision Diagram (BDD) which represents the ST-system underlying a high-level model such as a Petri net, UML state chart, set of reactive modules, etc.. The proposed scheme is independent of the model description method, as it combines state enumeration with symbolic, i. e., BDD-based, state space construction techniques. In contrast to existing techniques it exploits state-counter and activity-driven decomposition of high-level models, as this severely reduces the number of state enumerations and softens the hill climbing problem commonly attached to the incremental construction of BDDs. As shown in an empirical evaluation, the proposed technique has the potential not only for outperforming other combined approaches, but is also competitive w. r. t. other fully symbolic state space construction techniques.

1 Introduction

Motivation. For coping with the complexity of today's systems it is much more convenient and definitely less error-prone to describe systems by means of structured high-level modeling formalisms such as (extended) Petri Nets (PN), UML state chart, communicating processes, reactive modules, etc. , rather than directly capturing their behaviors in finite labeled transition systems (*LTS*). However, given that the *LTS* is the mathematical object to be studied, it is straightforward that one must somehow expand any high-level model into its underlying

LTS. This is commonly done by applying a modeling method's operational semantics in a fixed point computation. This computation which is known as state space exploration enumerates all states of a model's underlying *LTS* in an explicit manner. The number of states in the *LTS* is exponentially bound by the number of concurrent activities of the high-level model. This effect which is well known as state space explosion may hamper system analysis or even making it infeasible due to runtime and memory consumption of explicit state enumeration. In such a context Binary Decision Diagram (BDD) based techniques have shown to be very helpful for efficiently constructing and compactly representing high-level model's underlying *LTS*. Tools successfully employing such techniques range from the pure functional setting, e. g. NumSMV [1] up to stochastic model checkers like PRISM [21]. However, when it comes to tools like the Möbius performance analysis framework [19] which support various modeling, input languages resp., it is essential that Decision Diagram (DD) based state space construction is independent of the tool's modeling/input language. Independence in this respect can be achieved by carrying out enumeration and individual encoding of system states. This yields traditional state space exploration coupled with BDDs for storing states and state-to-state transitions. However, the individual and explicit processing of system states (and transitions) is extremely computationally expensive: (a) one deals with more than hundreds of billions of states and transitions, s.t. their individual generation and insertion imposes a non-acceptable run-time overhead. (b) the incrementally constructed BDDs for representing a model's set of reachable states or *LTS* severely suffers from the hill climbing problem.[1] For coping with these problems this paper introduces a new scheme for efficiently constructing a BDD-based representation of a high-level model's underlying *LTS* and its set of reachable states.

Organization. In the context of state-based system verification Decision Diagrams (DD) have been successfully employed for almost two decades. For taking this into account Sec. 2 carries out a classification of existing DD-based state space exploration techniques, which ultimately allows us to classify and pinpointing the contribution of this paper. Sec. 3 defines basics of high-level model description techniques and *ST*-systems as exploited in this work. It also introduces zero-suppressed Multi-terminal BDDs (ZBDDs) and shows how they can be employed for representing *ST*-systems. This prepares the ground for the new scheme which is presented in Sec. 4. Its empirical evaluation is carried out in Sec. 5. Sec. 6 concludes the paper.

2 Symbolic methods and own contribution

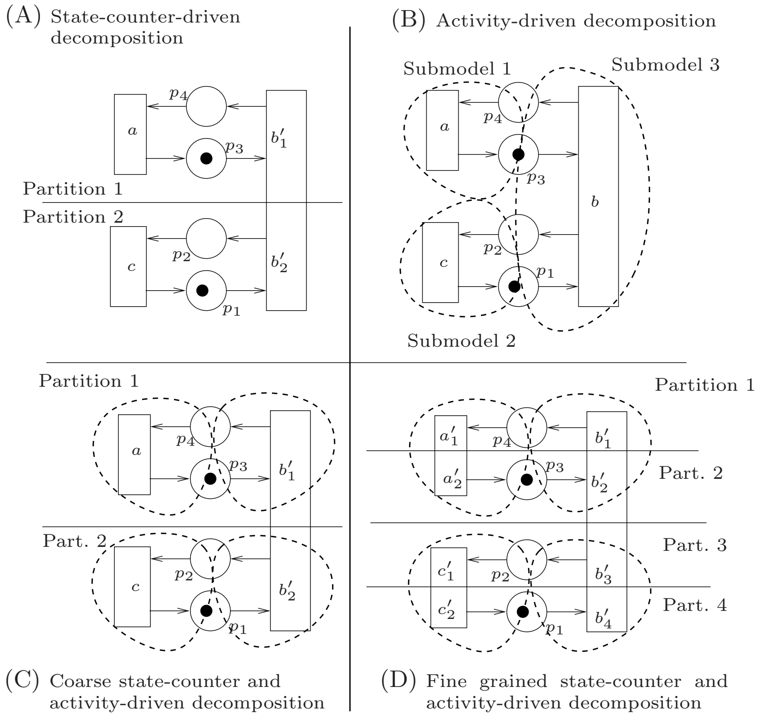
Classification. When it comes to BDD-based state space exploration one may distinguish among the following strategies:

[1] The hill climbing problem can be characterized as follows: The size of the BDD increases dramatically during the incremental insertion of states or transitions. However, towards the end of state space generation the BDDs are most likely to collapse and become very compact.

(1) *Pseudo-symbolic methods* exploit exhaustive state enumeration and BDD-based representation of transitions. As the detected transitions are individually inserted into a BDD representing the models underlying *LTS*, methods of this class, e. g. [8], severely suffer from run-time overheads and the hill climbing problem.
(2) *Semi-symbolic methods* execute explicit state space exploration and BDD-based encoding in a partial manner. For obtaining a model's complete *LTS*, BDD-based composition schemes apply, e. g. the one of Eq. 4. As symbolic composition constructs supersets of transitions semi-symbolic techniques rely on (partitioned) symbolic reachability analysis [4] or Ciardo's saturation method [5] for reducing the set of states and transitions to the actually reachable elements.
(3) *Fully symbolic methods* implement the transition rules of high-level modelling methods with generic BDD-algorithms. Hence these methods solely operate on the level of the symbolic data structures, where DD-based reachability schemes are employed for obtaining a model's complete *LTS*. The fully symbolic methods are very efficient, but contrary to the other techniques, they are restricted to (tool-) specific modeling methods, i. e., to those modeling languages which possess a BDD-based implementation of their operational semantics.

As pointed out above, individual exploration and encoding of states and transitions is costly. Hence contemporary BDD-based techniques intend to keep this at a moderate level. To do so one commonly exploits some form of compositionality allowing to generate a BDD-based representation of a model's overall *LTS* from smaller components in a symbolic composition scheme. However, as symbolic composition delivers a model's potential *LTS* it enforces the execution of a symbolic reachability scheme. For achieving compositionality the following strategies are known:

(1) *State-counter-driven decomposition of models*: The techniques of this class exploit activity-synchronization for constructing complex models from smaller components. This requires either a compositional modeling method in the style of a process algebra or a decomposition of the overall model. Such a decomposition in case of a Petri net (PN) is depicted in Fig. 1.A. As with this approach the state counters, here the places of the PN, are partitioned, activities manipulating state counters of different partitions have to be split accordingly (cf. Sec. 4.2). For obtaining the *LTS* of the overall model synchronization over (the previously split) activities needs to be implemented, but on the level of symbolic data structures [10,9,22]. The required cross-product computation is achieved by making use of Bryant's BDD-algorithms [2] or derivatives thereof. The usage of these algorithms make the BDD-based schemes more flexible than the original Kronecker-operator based procedure [20,7,6].
(2) *Activity driven decomposition of models*: The activity-local scheme [17] encapsulates each activity in its own submodel (cf. Fig. 1.B) and generates a(n) (activity-local) BDD for each of these submodels by enumerating states



(A) State-counter-driven decomposition

(B) Activity-driven decomposition

(C) Coarse state-counter and activity-driven decomposition

(D) Fine grained state-counter and activity-driven decomposition

Figure 1. Decomposition of high-level models

and encoding detected transitions.[2] Once a fixed point is reached, i. e., the activity-local BDDs are constructed, a symbolic composition scheme and symbolic reachability analysis is executed which deliver the overall *LTS* of a high-level model. With the activity-local procedure inefficiencies may occur, if the individual activities contained in a high-level model have large sets of input and output variables. In Fig. 1.B. submodel 3 would be such a case as activity b manipulates all 4 state counters.

Contribution. This paper presents a scheme which combines standard state enumeration with BDD-based techniques for constructing a high-level model's underlying *LTS* and its set of reachable states (= reachability set), BDD-based representations thereof resp.. In contrast to existing techniques this new scheme exploits *state-counter and activity-driven decomposition* for constructing the symbolic representations. In a nutshell, the proposed procedure requires a model's partitioning and applies the activity-local scheme [17], but now in a partition-wise manner. The partitioning could either be provided by the user, e. g. as

[2] We differ between submodels and partitions, since in case of the state-counter-driven decomposition the sets of partition-local elements which are state counters and (sub-)activities, are pairwise disjoint. With the activity-driven decomposition this is not the case, as the state counters are shared among the submodels.

illustrated in Fig. 1.C, or automatically generated by encapsulating each state counter in its own submodel as illustrated in Fig. 1.D. Overall this two-fold decomposition limits explicit state enumeration and transition encoding to a small portion of a high-level model's underlying *LTS*. As demonstrated in Sec. 5 the scheme therefore outperforms our previously developed activity-local scheme [17]. It also appears to be competitive w. r. t. fully symbolic approaches. However, contrary to them the presented scheme is generic, i. e., it is independent of a modelling tool's input language. One may also note that in our approach, we do not impose the partitions to possess any dedicated structural properties. The only thing which matters is that there is a dependency relation defined on a high-level model's set of state counters and activities.

3 Background Theory

By leaving (partial) state enumeration to the resp. modelling tool, the here presented scheme allows to treat high-level modelling methods/formalisms (almost) in a black-box manner. Hence we do not need to worry in this section about the operational semantics of high-level formalisms. We solely need to consider some basic properties derived from the structure of a high-level model.

3.1 High-level models

We define a high-level model as a quadruple $(\mathfrak{S}, \boldsymbol{s}^{\,\epsilon}, \mathcal{A}ct, \mathcal{C}on)$, with

- $\mathfrak{S} := \{\mathfrak{s}_1, \ldots, \mathfrak{s}_n\}$ as an ordered set of state variables (SV). As each SV can take values from a finite domain the states of a high-level model can be written as a vector of integers $\boldsymbol{s} \in \mathbb{S} \subset \mathbb{N}_0^{|\mathfrak{S}|}$.
- $\boldsymbol{s}^{\,\epsilon}$ is the high-level model's initial state, i. e., it provides the initial value assignment for the SVs.
- $\mathcal{A}ct := \{n, l, \ldots, z\}$ refers to the set of activities, the execution of which allows the model to evolve from state to state.
- With $\mathcal{C}on \subseteq \mathfrak{S} \times \mathcal{A}ct \cup \mathcal{A}ct \times \mathfrak{S}$ we address a connection relation, where we define a SV $\mathfrak{s}_i$ and an activity l as connected *iff* $\mathfrak{s}_i$ either changes its value once activity l is executed or the current value of $\mathfrak{s}_i$ influences the behavior of l.

With the connection relation $\mathcal{C}on$ one is enabled to define a set of dependent SVs for each activity l:

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

With $\mathfrak{S}_l^I = \mathfrak{S} \setminus \mathfrak{S}_l^D$ we address the set of independent SVs. These SVs are the ones which neither manipulate the behavior of activity l nor do they change their values once l is executed. Based on the above definition we define a projection function $\chi_l : \ \mathbb{N}_0^{|\mathfrak{S}|} \longrightarrow \mathbb{N}_0^{|\mathfrak{S}_l^D|}$ for each activity $l \in \mathcal{A}ct$. This function extracts the sub-vector w. r. t. activity l's set of dependent SVs and w. r. t. a state $\boldsymbol{s}$, where for simplicity the shorthand notation: $\boldsymbol{s}_{d_l} := \chi_l(\boldsymbol{s}\,)$ is employed by us.

The partial state vector $\boldsymbol{s}_{d_l}$ is called the activity-local marking of state $\boldsymbol{s}$ w. r. t. to activity l. The above definition of dependent SVs enables one furthermore to define a reflexive and symmetric dependency relation $\mathcal{A}ct^{\mathcal{D}} \subseteq \mathcal{A}ct \times \mathcal{A}ct$ where:

$$(k, l) \in \mathcal{A}ct^{\mathcal{D}} \Leftrightarrow \mathfrak{S}_k^D \cap \mathfrak{S}_l^D \neq \emptyset. \tag{1}$$

According to this, two activities $l, k \in \mathcal{A}ct$ are called dependent if they have at least one SV in common. In total this gives one a set of dependent activities for each activity l:

$$\mathcal{A}ct_l^D := \{k \in \mathcal{A}ct \mid (l, k) \in \mathcal{A}ct^{\mathcal{D}}\}. \tag{2}$$

Please note that also $l \in \mathcal{A}ct_l^D$ holds.

The above sets are important for keeping state space exploration partial, since they allow to execute a selective breadth-first-search scheme, rather than exhaustively enumerating and encoding the states of a models underlying *LTS*.

Example: As an example one may refer to Fig. 1.A. If we assume that the number of tokens contained in place i is recorded by SV $\mathfrak{s}_i$ the above discussion yields that activity a takes $\mathfrak{s}_1$ and $\mathfrak{s}_2$ as dependent SVs ($\mathfrak{S}_a^D = \{\mathfrak{s}_1, \mathfrak{s}_2\}$), whereas $\mathfrak{s}_3$ and $\mathfrak{s}_4$ are on its set of independent variables ($\mathfrak{S}_a^I = \{\mathfrak{s}_3, \mathfrak{s}_4\}$). The set of dependent activities of activity a is then defined as $\mathcal{A}ct_a^D := \{a, b\}$.

3.2 Low-level models

The operational semantics of a modelling formalism is irrelevant for the discussion to follow. This is because the proposed method relies on (partial) standard state space exploration, which makes it applicable for any kind of (state-based) high-level modelling formalism. The only thing which matters is that for a given source state $\boldsymbol{s}$ and a given activity l one is enabled to decide whether l is executable or not and if so that there is a method which returns the respective target state $\boldsymbol{t} \in \mathbb{N}_0^{|\mathfrak{S}|}$. How the latter is actually computed is irrelevant, we only require that this computation depends on the dependent SVs of the respective activity. For making this setting accessible we give the following informal definitions.

If for a model state $\boldsymbol{s}$ activity l is executable we say that l is enabled w. r. t. $\boldsymbol{s}$ and write $\boldsymbol{s}\ [> l$. When executing an enabled activity the high-level model evolves from one state to the next by executing the activity's transition function $\delta_l : \mathbb{N}_0^{|\mathfrak{S}|} \rightarrow \mathbb{N}_0^{|\mathfrak{S}|}$ which provides the target state $\boldsymbol{t}$ w. r. t. a source state $\boldsymbol{s}$:

$$\delta_l(\boldsymbol{s}) := \begin{cases} \boldsymbol{t} \Leftrightarrow \ \boldsymbol{s}\ [> l \\ \perp \text{else} \end{cases}$$

One may note that δ_l is assumed to solely depend and manipulate the positions in a state vector which refer to the SVs of $\mathfrak{S}_l^D$. The utilization of all δ_l-functions in a fixed point computation allows one to construct a *LTS* $T \subseteq (\mathbb{S} \times \mathcal{A}ct \times \mathbb{S})$ for a given high-level model, where $\mathbb{S} \subseteq \mathbb{N}_0^{|\mathfrak{S}|}$ is the model's set of reachable states. In the following target states of transitions may carry superscripts which refer to the sequence of activities the execution of which brought the resp. state about,

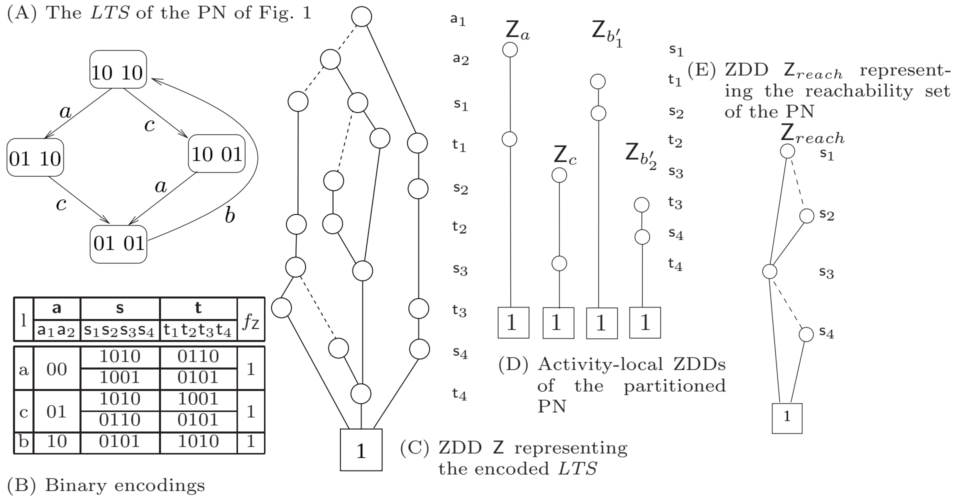
e. g. for $\boldsymbol{s}^{\,\omega} \in \mathbb{S}$ with $\omega := (\alpha, \ldots, \zeta) \in \mathcal{A}ct^*$ the state descriptor $\boldsymbol{s}^{\,\omega}$ refers to the activity execution sequence $\alpha, .., \zeta$ with $\boldsymbol{s}^{\,\omega} := \delta_\zeta(..\delta_\alpha(\boldsymbol{s}^{\,\epsilon})..)$ and where $\boldsymbol{s}^{\,\epsilon}$ is the high-level model's initial state.

Example: In case of the PN depicted in Fig. 1 each state can be represented by a vector with 4 elements, each indicating the number of tokens contained in a specific place. For the order $\mathfrak{s}_1 \prec \mathfrak{s}_2 \prec \mathfrak{s}_3 \prec \mathfrak{s}_4$ the initial state $\boldsymbol{s}^{\,\epsilon}$ is given by $(1, 0, 1, 0)$. It allows the execution of activity a and c, yielding the successor states $(0, 1, 1, 0)$ and $(1, 0, 0, 1)$. The *LTS* as obtained by carrying out standard state space generation is depicted in Fig. 2.B.

3.3 Symbolically representing *LTS*

In this paper we make use of **zero-suppressed** Multi-terminal Binary Decision Diagrams (ZDDs) [18]. We do so because for symbolic analysis of high-level models this relatively new type of decision diagram has shown its superiority if compared to standard BDDs [2] and their multi-terminal extensions, cf. [17,18].

The ZDD data structure and its algorithms ZDDs are directed acyclic graphs with a dedicated root node. If they are ordered and reduced, they allow (weakly) canonical representations of pseudo-Boolean function, i. e., of functions of the kind $f : \mathbb{B}^{|\mathcal{V}|} \to \mathbb{D}$, with $\mathcal{V} := \{\mathsf{v}_1, \ldots, \mathsf{v}_n\}$ as finite set of (Boolean) input or function variables and $\mathbb{D}$ as finite domain of function values, e. g. $\mathbb{B}$. For keeping ZDDs compact, isomorphic (sub)-structures are collapsed. As common each inner or non-terminal node of a ZDD is associated with an input variable of the represented function. In a ZDD the skipping of a variable implies that this variable is 0-assigned, and one denotes such a variable as 0-suppressed (*0-sup.*). In case of a BDD a skipping implies a 0 **or** 1 assignment of the respective variable and one speaks of a *don't care* variable. This slight difference of BDDs and ZDDs w. r. t. skipped variables has a severe effect: With multi-rooted DD as incorporated into packages like CUDD [23], ZDD-nodes loose their uniqueness as soon as the represented functions have differing sets of input variables. This destroys the efficiency of standard ZDD manipulations. To solve this problem [18] introduced the concept of partially shared ZDDs and the `ZApply` algorithm for applying binary operations to ZDDs. The `ZApply` is a extension of Bryant's famous `Apply`-algorithm, it features the following idea: 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 the standard *don't care*-interpretation in case of a non-input variable and Minato's *0-sup.*-interpretation in case of a skipped node which would have been labelled with an input variable. Boolean, set operators resp. and arithmetic operators can be applied to ZDDs on the basis of the `ZApply`-algorithm there these operators only differ in the handling of the terminal nodes. In the following we exploit the following dualities $+ = \vee = \cup$ and $\times = \wedge = \cap$ in our notation and write Z_a `op` Z_b when applying operation `op` $\in \{+, \times, -\}$ to the ZDDs Z_a and Z_b.



(A) The *LTS* of the PN of Fig. 1

l	a	s	t	f_Z
	a_1a_2	$s_1s_2s_3s_4$	$t_1t_2t_3t_4$	
a	00	1010	0110	1
		1001	0101	
c	01	1010	1001	1
		0110	0101	
b	10	0101	1010	1

(B) Binary encodings

(C) ZDD Z representing the encoded *LTS*

(D) Activity-local ZDDs of the partitioned PN

(E) ZDD Z_{reach} representing the reachability set of the PN

Figure 2. From a PN to the ZDD-based representation of its *LTS*

In the following we exploit the ZDD-operator `Execute`. It allows to execute a symbolically represented transition function e. g. the one implementing function δ_l, w. r. t. a symbolically represented set of states in a single operation. However, contrary to the original ideas of relational products [3] algorithm `Execute` handles transition relations which are not defined on the complete set of s-variables. This is achieved by employing an identity semantics to pairs of non-essential s and t variables. In the following we also employ the algorithm $\texttt{Abstract}(+, S, \mathsf{Z})$ which allows the existential quantification of a ZDD Z w. r. t. the variables of set S , i. e., we compute the ZDD $\mathsf{Z}' := \sum_{\forall x \in S} \mathsf{Z}_{x=0} + \mathsf{Z}_{x=1}$, where the variables of S are not essential.

Example: An exemplification of a ZDD is provided by Fig. 2.C. A dashed (solid) lines in the ZDD indicate the value assignment 0 (1) to the corresponding Boolean variable on the respective path. The ZDD is ordered, i. e., we have the same variable ordering one each path. As the order is from top to bottom, we could safely omit the arrow heads on the node connecting edges. For clarity we also omitted the terminal 0-node and its incoming edges. The ZDD is reduced, i. e., we do not show isomorphic nodes and we do not show nodes the outgoing 1-edge of which leads to the terminal 0-node, e. g. in the outer left path the variable t_1, s_2 and s_4 and t_4 are skipped.
Contrary to sub-figure (C) the ZDDs of sub-figure (D) are assumed to symbolically represented functions which do not take all variables as input. Let Z_a only take the variables $\{s_1, \ldots, t_2\}$ as input, s.t. the variables $\{s_3, \ldots, t_4\}$ are non-input variables, i. e., they are not essential. In such a setting Z_a represents the Boolean function $f(s_1, t_1, s_2, t_2) := s_1 \times (\neg t_1) \times (\neg s_2) \times t_2$.

ZDD-based representation of *LTS* Explicit enumeration of all a high-level model's states and state-to-state transitions allows to construct an *LTS* $T \subseteq$

$(\mathbb{S} \times \mathcal{A}ct \times \mathbb{S})$. Each transition of such a *LTS* T can now be encoded by applying a binary encoding function `Encode` which transforms each labeled transition $(\boldsymbol{s} \overset{l}{\longrightarrow} \boldsymbol{s}^{\,l})$ into a bit-vector. The individual bit positions correspond to the input variables of the ZDD, where we have the convention that a-variables hold the binary encoded activity label, s-variables the encodings of the source and t-variables the encodings of target states. As common the variables are ordered in an interleaved fashion: $\mathsf{a}_1 \prec \ldots \prec \mathsf{a}_{n_{\mathcal{A}ct}} \prec \mathsf{s}_1 \prec \mathsf{t}_1 \prec \ldots \prec \mathsf{s}_n \prec \mathsf{t}_n$. This encoding yields a function table constructible for each finite *LTS*, where functions of this kind can canonically be represented by partially shared ZDDs.

Example: Fig. 2.A - C sketches an example: Table B shows the (binary) encodings of the *LTS* depicted in Fig. 2.A, constituting the function table of a Boolean function. The ZDD constructible from this function table is shown in Fig. 2.C. E.g. the transition $(1,0,1,0) \overset{a}{\to} (0,1,1,0)$ the encoding of which is depicted in line 1 of Table 2.B is represented by the outer left path in the ZDD of Fig. 2.C, please remember the interleaved ordering of s and t variables.

4 Combined scheme for constructing *LTS*

As main goal we hope to limit the number of explicit state enumerations and individual encoding of transitions as far as possible, as this is computationally expensive. The vast majority of transitions will be obtained by BDD-based computations which construct all possible interleaved transition executions by cross-product computations.

4.1 Preliminaries

At first there is a need to clarify some notation.

Dependency sets. For convenience we introduce the following sets:

$$\mathsf{D}_l := \{\mathbf{s}^{\,i}, \mathbf{t}^{\,i} | \mathfrak{s}_i \in \mathfrak{S}_l^D\} \text{ and } \mathsf{I}_l := \{\mathbf{s}^{\,i}, \mathbf{t}^{\,i} | \mathfrak{s}_i \in \mathfrak{S}_l^I\}, \qquad (3)$$

where $\mathbf{s}^{\,i}$ and $\mathbf{t}^{\,i}$ refer to those Boolean variables which encode the value of dependent SV $\mathfrak{s}_i$ in the source and target state of a transition with respect to activity l. Consequently the set I_l refers to l's set of independent SVs, i.e. their Boolean counterparts, respectively. In case it is required we will make use of the symbols $\mathsf{I}_l^{\mathsf{s}}, \mathsf{D}_l^{\mathsf{s}}$ and $\mathsf{I}_l^{\mathsf{t}}, \mathsf{D}_l^{\mathsf{t}}$ when referring to the sets restricted to the s- or t-variables.

Activity-local ZDDs. The scheme to be presented targets the generation of activity-local transition function, one for each activity of the high-level model. As these ZDDs may take only subsets of variables as input, we employ the notation $\mathsf{Z}_k{<}a,b,c{>}$ for indicating that ZDD Z_k which refers to the transitions induced by activity k only tales the variables a, b and c as input.

Symbolic composition. For obtaining a symbolic representation of the model's potential *LTS* DD-based schemes traditionally compute supersets of transitions

[10,9,22]. The obtained potential transition function can be employed in a symbolic reachability analysis for constructing thw ZDD Z_{reach} which is the symbolic representation of a model's set of reachable states. Given the set of activity-local ZDDs, we can construct the overall *LTS* of the high-level model as follows:

$$\sum_{k\in\mathcal{A}ct\setminus\mathcal{A}ct_{\mathcal{S}}} \mathsf{Z}_k \times \bot <\mathsf{I}_l> \ + \sum_{l\in\mathcal{A}ct_{\mathcal{S}}} \left(\prod_{\forall l'_i \in l} \mathsf{Z}_{l'_i} \right) \times \bot <\mathsf{I}_l> \tag{4}$$

where $\bot <\mathsf{I}_l>$ is an identity structure over the set of activity l's set of independent SVs, their boolean counter parts respectively (cf. definition below). The above equation basically follows the Kronecker-operator-based approach of [20,7,6]. However, as the BDD-operators can cope with partition-wise nested variable orderings, the composition scheme of BDDs is much more flexible. The insertion of identity structures has to do with the circumstance that the resp. variables maintain their values once the resp. activity is executed.

4.2 The scheme at glance

In a nutshell, we propose a new technique, which depends (a) on model decomposition, (b) on partial explicit exploration and individual encoding of transitions, and (c) on pure symbolic manipulations for obtaining the set of reachable states and transitions of a high-level model.

(A) State-counter-driven decomposition It is required that the overall model exhibits some compositional structure. State-counter-driven decomposition addresses the partitioning of a model into n parts, where *activities manipulating SVs of more than one partition are split accordingly*, s.t. $\forall i,j \in \{1,\ldots,n\}$ with $i \neq j : \mathfrak{S}^i \cap \mathfrak{S}^j = \emptyset$ holds. One may note that in our approach, we do not impose any structural properties on the partitions. The presented approach works with an arbitrary partitioning provided by the user or by automatically encapsulating each SV in its own partition and split the connecting activities accordingly. For obtaining the model's overall *LTS* from the partition-local *LTS*, synchronization of the previously split activities takes place. We indicate the splitting of an activity by priming and indexing the resulting (sub-)activities accordingly. The set of activities to be split is denoted $\mathcal{A}ct_{\mathcal{S}}$. One may note that $\mathcal{A}ct_{\mathcal{S}}$ can be interpreted as multi-set, as each of its element's refers to a set of sub-activities to be synchronized (see FOR-loop in Algo. 3.D (line 7-11)). The definitions of Sec. 3.1 can now be extended to the partition-local case: each partition $\mathcal{P}_i$ consists of a finite ordered set of discrete SVs $\mathfrak{s}^i_j \in \mathfrak{S}^i$, and a finite set of activities ($\mathcal{A}ct^i$) with a connection relation among them. 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 over these SVs. Since the sets of partition-local activities, as well as their sets of SVs are pairwise disjoint the dependency relation defined in Sec. 3.1 extends also to the partition-local case. An exemplification is provided by Fig. 1: with our approach state counter driven decomposition of the PN is allowed to deliver an arbitrary partitioning, e. g. one may choose a decomposition into two partitions as illustrated in Fig. 1.C. Partition $\mathcal{P}_1$ contains then the places p_1 and p_2 and partition $\mathcal{P}_2$ the remaining places p_3 and p_4. Activity b has to be split into the two sub-activities b'_1 and b'_2, s.t. the defined model components are truly disjoint. Alternatively we could automatically decompose the PN into 4 partitions, one for every SV and with each sub-activity as submodel for the activity-local scheme (cf.Fig. 1.D).

(B) Generating and encoding partition/activity-local transition systems Once a high-level model is decomposed, the next step deals with the generation of symbolic representations. Applying the activity-local scheme for each partition, generates a number of transition functions, their ZDD-based representations resp.. We obtain one ZDD for each (non-split) activity, denoted ZDD Z_l, and one for each sub-activity l'_i, denoted ZDD $\mathsf{Z}_{l'_i}$.

(C) Symbolic manipulations for the reachability set In our approach symbolic composition can be omitted. This is possible for the following reasons: (a) we employ the ZDD-operator `Execute` [16] within the symbolic reachability analysis (routine `SymbReach` (Algo. 3.D)). This operator executes a symbolic image computation w. r. t. partial transition functions and for ZDD-based state representations. (b) instead of applying a precomputed cross-product for each set of synchronizing activities we employ the transition functions one by one before updating the set of newly reached states (cf. Algo. 3.D (line 9 and 10)). This strategy is based on the associativity of multiplication and in principle implements the product employed in Eq. 4.
Once symbolic reachability analysis reaches a global fixed point a ZDD Z_{reach} representing a high-level model's reachability set is constructed.

In the following we go through the algorithms presented in Fig. 3 and implementing the above steps.

4.3 Implementation details

In lines 1 - 4 of the top-level algorithm (Fig. 3.A) some data initialization is done: ZDD Z_{reach} is set to the initial state $\boldsymbol{s}^{\epsilon}$ and the partition-local buffers StateBuffer and TransBuffer are allocated. The entity $\mathsf{StateBuffer}_i$ is used for holding tuples of states and activities, where the activities are supposed to be executed in the state. Entity $\mathsf{TransBuffer}_i$ holds transitions to be encoded and inserted into the respective activity-local ZDD. Routine `Initialize`() fills $\mathsf{StateBuffer}_i$ with the (initial) elements to be explored, i. e., with tuples consisting of the initial state and an activity to be executed in this state, more details on (re)-initialization follow.

Partition-wise generation of transitions As main feature we generate a partition-local transition function Z_l for each partition-local activity l, including the sub-activities l'_i. To do so routine `ExplorePartition`() (Algo. 3.C) executes explicit exploration of states (line 5) and encodes each of the detected partition-local transitions (line 9). This is repeated until no new transition can be detected or a maximum number of explicit state exploration steps has been executed. This latter maximum is necessary, as the partition-local transition system may not be

(A) Main routine

```
ConstructLTS()
(1)  $Z_{reach} := \mathsf{Encode}(\boldsymbol{s}^{\epsilon})$
(2)  $\forall i : \mathsf{StateBuffer}_i := empty$
(3)  $\forall i : \mathsf{TransBuffer}_i := empty$
(4)  $\forall i$ : Initialize($i$)
(5)  WHILE $\exists i : \mathsf{StateBuffer}_i \neq \emptyset$ DO
(6)     $\forall i$ : ExplorePartition($i$)
(7)     $Z_{reach} := Z_{reach} + $ SymbReach()
(8)     $\forall i$ : Initialize($i$)
```

(B) (Re-)initializing exploration

```
Initialize($i$)
(1) FOR $l \in \mathcal{A}ct^i$ DO
(2)   $Z_{new} := Z_{reach} - \mathsf{E}^i_l$
(3)   WHILE $Z_{new} \neq \emptyset$ DO
(4)      $Z_s := \mathsf{ExtractState}(Z_{new})$
(5)      $\boldsymbol{s} := \mathsf{Encode}^{-1}(Z_s)$
(6)      IF $\boldsymbol{s}\ [> l$ THEN
            $\mathsf{push}(\mathsf{StateBuffer}_i, (\boldsymbol{s}, \{l\}))$
(7)      ENDIF
(8)      $Z_{new} := Z_{new} - $ Abstract$(+, \mathsf{I}^{\mathsf{s}}_l, Z_s)$
```

(C) Activity-local scheme for exploring a partition

```
ExplorePartition($i$)
(1)  DO{$cnt := cnt + 1$
(2)       WHILE $\mathsf{StateBuffer}_i \neq empty$ DO
(3)          $\mathsf{pop}((\boldsymbol{s}, \mathcal{F}^l_{\boldsymbol{s}}), \mathsf{StateBuffer}_i)$
(4)          FOR $k \in \mathcal{F}^l_{\boldsymbol{s}}$ DO
(5)             $\boldsymbol{s}^{\,k} := \delta_k(\boldsymbol{s})$
(6)             $\mathsf{push}(\mathsf{TransBuffer}_i, (\boldsymbol{s}, k, \boldsymbol{s}^{\,k}))$
(7)       WHILE $\mathsf{TransBuffer}_i \neq empty$ DO
(8)          $\mathsf{pop}(\mathsf{TransBuffer}_i, (\boldsymbol{s}, l, \boldsymbol{s}^{\,l}))$
(9)          $Z_l := Z_l + \mathsf{Encode}(\boldsymbol{s}_{d_l}, \boldsymbol{s}^{\,l}_{d_l})$
(10)         IF $cnt \leq MAX$ THEN
(11)            $\mathcal{F}^l_{\boldsymbol{s}^{\,l}} := \emptyset$
(12)            FOR $k \in \mathcal{A}ct^D_l$ DO
(13)               IF $(\mathsf{Encode}(\boldsymbol{s}^{\,l}_{d_k}) \times \mathsf{E}^i_k = 0) \wedge$
                                      $\boldsymbol{s}^{\,l}\ [> k$ THEN
(14)                  $\mathcal{F}^l_{\boldsymbol{s}^{\,l}} := \mathcal{F}^l_{\boldsymbol{s}^{\,l}} \cup \{k\}$
(15)                  $\mathsf{E}^i_k := \mathsf{E}^i_k + \boldsymbol{s}^{\,l}_{d_k}$
(16)               ENDIF
(17)            IF $\mathcal{F}^l_{\boldsymbol{s}^{\,l}} \neq \emptyset$ THEN
(18)               $\mathsf{push}(\mathsf{StateBuffer}_i, (\boldsymbol{s}^{\,l}, \mathcal{F}^l_{\boldsymbol{s}^{\,l}}))$
(19)            ENDIF
(20)         ENDIF
(21) } WHILE $\mathsf{StateBuffer}_i \neq \emptyset \wedge cnt \leq MAX$
```

(D) Symbolic Reachability analysis

```
SymbReach()
(1)  $Z_{unex} := Z_{reach}$
(2)  DO{
(3)        $Z_{reach} := Z_{unex} + Z_{reach}$
/* Execute non-synch. activities */
(4)        FOR $l \in \bigcup_{\forall i} \mathcal{A}ct^i \setminus \mathcal{A}ct_{\mathcal{S}}$ DO
(5)              $Z_{new} := \mathsf{Execute}(Z_{unex}, Z_l)$
(6)              $Z_{unex} := Z_{unex} + Z_{new}$
/* Execute synch. activities */
(7)        FOR $l \in \mathcal{A}ct_{\mathcal{S}}$ DO
(8)           $Z_{new} := Z_{unex}$
(9)           FOR $l'_k \in l$ DO
(10)             $Z_{new} := \mathsf{Execute}(Z_{new}, Z_{l'_k})$
(11)          $Z_{unex} := Z_{unex} + Z_{new}$
(12)       $Z_{unex} := Z_{unex} - Z_{reach}$
(13) } WHILE $Z_{unex} \neq \emptyset$ DO
(14) RETURN $Z_{reach}$
```

Figure 3. Algorithms for the new scheme

finite when explored in isolation. The most distinguished feature of this approach is the *selective* bfs exploration of states. A selective bfs exploration scheme is obtained by only executing activity k in a state $\boldsymbol{s}^{\,\omega l}$ *iff* activity k depends on the last activity the execution of which brought the state $\boldsymbol{s}^{\,\omega l}$ about (here l) and the activity-local marking of the current state $\boldsymbol{s}^{\,\omega l}_{d_k}$ has not been tested with activity k before ($k \in \mathcal{A}ct^D_l \wedge \boldsymbol{s}^{\,\omega l}_{d_k} \notin \mathsf{E}^i_k$, FOR-loop of Algo. 3.C (line 12 -18)). For simplicity we store the activity markings which already have been tested on activity k in a respective symbolic structure denoted E^k_l. One may note that this "partial-order-like" exploration scheme suffices, as sequences of independent activities are expanded when carrying out symbolic reachability analysis.[3] The above steps are implemented with the help of two complementary $while - loop$s contained in Algo. 3.C:

The upper loop fetches states and lists of activities from the local buffer $\mathsf{StateBuffer}_i$ (line 3) and computes for each activity from that list the successor state $\boldsymbol{s}^{\,k}$ (line 5); where the obtained transitions are inserted into the buffer $\mathsf{TransBuffer}_i$ (line 6). The lower $while - loop$ reads the individual transitions from that buffer and individually encodes them, allowing their insertion into the respective partition and activity-local ZDD (line 8 and 9). As long as the maximum number of state enumerations has not been reached (line 10), one computes the set of dependent activities enabled in the target state of the previously handled transition (line 12 - 19). The obtained set denoted $\mathcal{F}^l_{\boldsymbol{s}^{\,l}}$ is then together with the target

[3] Correctness and completeness of such a scheme was provided in [15], where the argumentation extends to the case of decomposed models accordingly.

state inserted into the buffer $\mathsf{StateBuffer}_i$ for explicit exploration (line 18). One may note that we only explore activities on states if the resp. activity was not already tested in that state and if the activity is enabled (line 13). As we also only test activities which are on the dependency set of the activity whose execution brought the currently considered target state $\boldsymbol{s}^{\,l}$ about, here l, we obtain the selective breadth-first search scheme already introduced above. Both loops of Algo. 3.C are executed alternately until one reaches a partition-local fixed point ($\mathsf{StateBuffer}_i = \emptyset$) or the maximum number of state enumerations has been reached. In the first case one has visited all partition-local states reachable from the initial state(s) through sequences of dependent activities. In the second case the maximum number of state enumerations has been reached, where we resume with state enumeration and transition encoding in the next round of partition-local explorations and only if re-initialization (`Initialize`) indicates the necessity of doing so. As already mentioned, this catches the case that a partition-local transition relation is not finite when considered in isolation. E.g. one may consider the partitioning provided in Fig. 1.D; where partition-local activity b'_1 can be executed infinitely often, as its pre-set of places is empty.

With `ExplorePartition` terminated that follows next, is the execution of routine `SymbReach` for obtaining a model's set of reachable states.

Symbolic reachability analysis We organize symbolic reachability analysis in a partitioned, quasi-depth-first search (qdfs) manner. In the upper loop algorithm `SymbReach`() executes symbolic reachability analysis for the partition-local activities which are not synchronizing (line 2-9 of Fig. 3.D). The lower loop (line 10-15) executes the symbolic reachability analysis for the synchronizing activities.In a nutshell, algorithm `SymbReach` applies the semantic of Eq. 4 and it does this as follows: ZDD-operator `Execute` computes the one-step reachability sets (image) for a ZDD-based set representation of states and a transition function which may take subsets of function variables as its input variables. This allows us to execute all transition functions referring to non-synchronizing activities (FOR-loop of line 4-6). The execution of synchronizing activities follows in the lower FOR-loop (line 7 -11) analogously. As we execute the synchronizing activities referring to the same label l sequentially before updating the set of newly reached states (line 9-11), we implement the cross-product of previously split activities.

The execution of the transition functions in the different loops may deliver new unexplored states. Therefore we must test this and re-enter the outer WHILE-loop if necessary. The execution of the latter is repeated until no new states can be detected and a fixed point is reached. As the constructed set of reachable states may include states which result from the *interleaved execution* of independent activities, these states must be tested if they trigger new partition-local behaviors. This is done with routine `Initialize`().

Re-initialization of the scheme As main feature routine `Initialize`() only tests states the partition- and activity-local markings of which have not already been tested with the respective activity (line 2 of Algo. 3.B)). If the current partition-local activity, e. g. activity l, is enables in such a state, routine `Initialize`() pushes the respective state and activity pair into the partition-local buffer $\mathsf{StateBuffer}_i$ for further exploration (line 6 and 7 of Algo. 3.B).

If Algo. 3.B does not find any of such pairs, i. e., $\forall i : \mathsf{StateBuffer}_i = \emptyset$ a global fixed point is reached and the overall scheme terminates.

Example We consider the partitioning illustrated in Fig. 1.C, i. e., $\mathcal{P}_1$ contains activity a, sub-activity b'_1, and their sets of input- and output places, here p_1 and p_2. When executed the scheme generates ZDD Z_a and $\mathsf{Z}_{b'_1}$ for partition 1 and Z_c and $\mathsf{Z}_{b'_2}$ for partition 2 (cf. Fig. 2.D). It also produces ZDD Z_{reach} which represents the PN's reachability set (cf. Fig. 2.E).
In case of $\mathcal{P}_1$ the selective bfs exploration works as follows: in routine `Initialize` one detects that activity a is enabled in state $\boldsymbol{s}^{\,\epsilon} := (1, 0, 1, 0)$ and that the activity a has not yet been tested on states carrying the value $(1, 0, *, *)$, where * means *dnc* (line 3-8 of routine `Initialize`). Exploring a in state $\boldsymbol{s}^{\,\epsilon}$ yields the transition: $(1, 0, 1, 0) \xrightarrow{a} (0, 1, 1, 0)$, which is symbolically represented by function $\mathsf{Z}_a(\mathsf{s}_1, \mathsf{t}_1, \mathsf{s}_2, \mathsf{t}_2) = \mathsf{s}_1 \neg \mathsf{t}_1 \neg \mathsf{s}_2 \mathsf{t}_2$ and inserted into the ZDDZ_a (cf. Fig. 2.D). The values referring to the variables $\{\mathsf{s}_3, \ldots, \mathsf{t}_4\}$ are simply ignored, as they are not on the input set of Z_a. Rather than executing now all enabled activities in the newly reached state $(0, 1, 1, 0)$ one does so only for those enabled activities which are on a's set of dependent activities (here a and b'_1) and which have not been tested on states containing the marking $(0, 1, *, *)$. Since these two conditions only hold for sub-activity b'_1, one solely explore this activity within the newly reached state $(0, 1, 1, 0)$. The encoding of the detected transition yields the ZDD $\mathsf{Z}_{b'_1}(\mathsf{s}_1, \mathsf{t}_1, \mathsf{s}_2, \mathsf{t}_2) = \neg \mathsf{s}_1 \mathsf{t}_1 \mathsf{s}_2 \neg \mathsf{t}_2$ (cf. Fig. 2.E). The above steps are repeated until all traces of dependent activities of partition 1 have been explored. Now one proceeds with partition $\mathcal{P}_2$ in the same way, which yields the ZDDs Z_c and $\mathsf{Z}_{b'_2}$ (cf. Fig. 2.D). Once all partitions have been explored, one executes the obtained transition functions in a symbolic reachability analysis which delivers here the ZDD Z_{reach} (cf. Fig. 2.E). As Z_{reach} may contain states which might trigger new partition-local behavior re-initialization follows. As routine `Initialize` does not detect such states here, a global fixed point is reached and the scheme terminates.

5 Benchmarking the new scheme

Remarks. We implemented our scheme within the Möbius performance analysis framework [19]. For benchmarking the following models have been chosen: the Flexible Manufacturing system (FMS) [7], the Cyclic Sever system (Polling) [12], the Kanban manufacturing model (Kanban) [6] and the Tandem Queueing model (Tandem) [11]. The experiments were executed on a Intel Duo Core platform (2 GHz), equipped with 1 GByte of RAM and a Linux OS. So far an automatic decomposition of models is not implemented yet. Hence at the current stage, we partition the high-level models manually.

Model features			Old act.-local scheme			New partitioned scheme			CASPA	
N	$\#states$	$\#trans.$	$\#trans_e$	$t_{generate}$	mem_{peak}	$\#trans_e$	$t_{generate}$	mem_{peak}	$t_{generate}$	mem_{peak}
FMS										
15	7.24E+008	7.38E+009	3,920	5.79	2,897,217	4,013	4.18	3,007,709	2037.83	4,723,381
20	8.83E+009	9.50E+010	8,260	21.6	6,672,758	8,383	13.62	7,902,461	*	*
Polling										
10	1.54E+004	8.96E+004	40	0.02	15,682	70	0.02	15,121	0.03	3,922
15	7.37E+005	1.58E+007	60	0.06	64,605	105	0.05	36,331	0.06	9,903
Kanban										
15	4.70E+010	6.13E+011	299,280	47.29	26,039,756	2,400	2.14	760,530	4.91	248,749
20	8.05E+011	1.10E+013	*	*	*	4,200	3.28	1,772,080	39.79	881,062
25	7.68E+012	1.08E+014	*	*	*	6,500	7.62	3,418,899	61.34	334,855
Tandem										
1024	2.10E+006	7.33E+006	2,096,127	225.66	58,176,776	8,187	27.08	313,260	0.92	18,725
2046	8.38E+006	2.93E+007	*	*	*	16,371	124.51	891,363	2.68	37,207

* indicates that we aborted the *LTS* generation due to memory overflows.

Table 1. Benchmarking the partitioned activity-local scheme (tools based on CUDD)

Comparison of BDD-based schemes. At first we compare the new scheme to the original, non-partitioned activity-local scheme [17]. The results of the comparison are shown in Tab. 1. In the first three columns we show the model's scaling parameter N and the size of its underlying *LTS*, i.e. number of states and transitions. The remaining columns contain the number of transitions explicitly generated and encoded ($trans_e$), the run-time $t_{generate}$ in sec. and the peak number of temporarily allocated ZDD-nodes (mem_{peak}) when generating the *LTS* of the models. As illustrated by the figures of Tab. 1 the new scheme shows not a significant improvement in case of the FMS and Polling model. This stems from a badly chosen partitioning of the overall model: for resolving arc-cardinalities we had to insert additional activities into the different partitions of the FMS model. Overall this leads to slightly larger values for $trans_e$ for the new scheme, which in total leads to larger memory consumptions. The decrease in run-time might be explainable by a more efficient symbolic reachability analysis, as we are executing the split activities individually, instead of using the product of their transition functions in a single step.

In case of the Polling model we also had to come up with a more complex model. The implementation of the new scheme requires the synchronization of previously split activities. This enforced the explicit modelling of the server which sequentially synchronizes with the client whose request is handled next. In the original model, as analyzed by us with the standard activity-local scheme, this is not necessary, as a single (server) token travels from client to client making the modelling of the polling server itself unnecessary. Overall the different models give that the new scheme produces here also slightly larger values for $trans_e$. However, from the obtained run-times we could not judge if the new scheme really outperforms the standard activity-local scheme; due to the complexity of modelling the server model we did also not scale the Polling model further up. When it comes to the analysis of structured models a different picture has to be drawn. In such cases the new scheme clearly outperforms the activity-local scheme. In case of Kanban ($N = 20, 25$) and TQN ($N = 1024, 2046$) the peak memory consumption of the activity-local scheme is even beyond 2 GByte which enforces the operating system to abort the process. This is much better with the new scheme: the number of the transitions to be explicitly explored and encoded (col. $trans_e$) is much lower, consequently one obtains much lower run-times and peak memory requirements.

For benchmarking the presented scheme with other BDD-based tools, a set of experiments with the stochastic model checker CASPA [14] was executed. We did not employ the well known model checker PRISM [21], as PRISM is not optimized w. r. t. state space exploration; the bottleneck of stochastic model checking is the numerical analysis. Like PRISM, the tool CASPA also employs the BDD-library CUDD [23] and executes a symbolic reachability analysis which is organized in a (disadvantageous) breadth-first search manner. As this is leads to a very bad performance, we included a partitioned symbolic reachability analysis into CASPA, thereby improving its run-times by up to two orders of magnitude. The obtained run-times and memory consumptions are shown in the last 2 columns of Tab. 1. The new scheme clearly outperforms CASPA in case of the FMS, Polling and Kanban model. The superior performance of the new scheme can be explained with the very low number of transitions to be explicitly generated and encoded, in particular for the Polling and Kanban model, which induces a small overhead only, so that almost all of CPU-time consumption and memory usage is induced by symbolic reachability analysis. However, as soon as the number of explicitly generated and encoded transitions increases, CPU-time consumption and memory usage shifts towards explicit generation. Consequently in such cases the fully symbolic method of CASPA which avoids explicit *LTS* generation and encoding is more efficient. This is documented by the figures related to the TQN model as given in Table 1. The difference in memory consumption can be explained by the fact, that CASPA uses standard BDDs, a denser encoding scheme for the individual states and a fixed ordering of the BDD-variables which may significantly differ form the ordering chosen for the new scheme.

Comparison to SMART. In Table 2 we compare our new scheme with the symbolic model checker SMART, where we employed the transition-oriented saturation scheme. With this option the transitions as induced by the individual activities of the high-level model are stored in separate Multi-valued DD (MDD) [13], one for each activity as in case of the here presented scheme. However, instead of using symbolic reachability analysis SMART executes the saturation technique [5]. We execute the set of experiments twice, namely one time with partitioned and one time with non-partitioned models, as this gives different results, especially for the Kanban and FMS model.In fact for the FMS model we

N	SMART				New partitioned scheme	
	partitioned		not partitioned			
	$t_{generate}$	mem_{peak}	$t_{generate}$	mem_{peak}	$t_{generate}$	mem_{peak}
FMS						
15	nf ($> 1.44 \cdot 10^5$)		0.2	11,658	4.18	3,007,709
20	nf		0.46	15,175	13.62	7,902,461
Polling						
10	0.02	1,130	0.04	1,635	0.02	15,121
15	0.04	1,600	0.06	2,573	0.05	36,331
Kanban						
15	2.34	120,500	0.32	20,889	2.14	760,530
20	8.47	284,351	0.82	43,453	3.28	1,772,080
25	25.39	531,352	1.94	70,748	7.62	3,418,899
Tandem						
1023	10.72	187,083	9.54	201,040	27.08	313,260
2046	59.18	395,053	54.25	407,068	124.51	891,363

nf indicates abortion of the process after an reasonable amount of time.

Table 2. Benchmarking the new scheme with the SMART model checker

were not able to find a good partitioning, s.t. SMART could generate the reachability set in reasonable time (we aborted the procedure after $1.44 \cdot 10^5$ sec.). Only for the partitioned Kanban model the here proposed technique seems to be competitive if compared to Ciardo's saturation technique. For all other cases SMART outperforms the here presented method. This might be due to the very high-level of maturity of SMART. However, one should remember that the here proposed method is independent of the modelling language and that we employed the generic BDD-package CUDD. Contrary to this, SMART operates with its native input language which is based on Petri nets and makes use of its trimmed, native implementation of MDDs.

6 Conclusion

This paper presents a new semi-symbolic technique for constructing a high-level model's reachability set as well as its underlying *LTS*. As key feature the presented approach appears to be independent of the modelling method, which makes it applicable for a wide range of tools. But this independence has its price, namely it comes with explicit enumeration and encoding of states and transitions. For keeping this overhead as low as possible the proposed technique exploits a state-counter driven partitioning of a high-level model and makes use of a selective breadth-first-search exploration scheme. As demonstrated by the benchmarking models the proposed scheme has the potential for outperforming existing techniques which also rely on state enumeration and symbolic state space construction techniques. When compared to tool-specific, fully symbolic technique it still achieves competitive results.

References

1. The NuSMV Model Checker. nusmv.irst.itc.it.
2. R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
3. J. R. Burch, E. M. Clarke, K. L. McMillan, and David L. Dill. Sequential circuit verification using symbolic model checking. In *DAC '90: Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 46–51, New York, NY, USA, 1990. ACM.
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, G. Lüttgen, and A. S. Miner. Exploiting interleaving semantics in symbolic state-space generation. *Form. Methods Syst. Des.*, 31(1):63–100, 2007.
6. 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.
7. G. Ciardo and K. Trivedi. A decomposition approach for stochastic reward net models. *Performance Evaluation*, 18(1):37–59, 1993.
8. 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.
9. 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 S. Graf and M. Schwartzbach, editors, *Proc. of TACAS'00*, LNCS 1785, pages 395–410, Berlin, 2000. Springer.
10. R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for symbolic model checking in CCS. *Distributed Computing*, 6(3):155–164, 1993.
11. 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.
12. O. Ibe and K. Trivedi. Stochastic Petri net models of polling systems. *IEEE Journal on Selected Areas in Communications*, 8(9):1649–1657, 1990.
13. 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.
14. 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.
15. K. Lampka. *A symbolic approach to the state graph based analysis of high-level Markov reward models.* PhD thesis, University of Erlangen-Nuremberg, Erlangen (Germany), 2007.
16. K. Lampka. A new algorithm for partitioned symbolic reachability analysis. *ENTCS 223*, Workshop on Reachability Problems, 2008.
17. K. Lampka and M. Siegle. Activity-Local State Graph Generation for High-Level Stochastic Models. In *MMB'06*, pages 245–264, 2006.
18. K. Lampka, M. Siegle, J. Ossowskis, and C. Baier. Partially-shared zero-suppressed Multi-Terminal BDDs: Concept, Algorithms and Applications, 2008. Accepted for the journal FMSD, ftp.tik.ee.ethz.ch/pub/publications/TIK-Report-289.pdf.
19. Möbius page. www.mobius.uiuc.edu.

20. 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.
21. PRISM web page. www.prismmodelchecker.org.
22. M. Siegle. Advances in model representation. In Luca de Alfaro and Stephen Gilmore, editors, *Process Algebra and Probabilistic Methods*, LNCS 2165, pages 1–22. Springer, 2001. Proc. of the Joint Int. Workshop, PAPM-PROBMIV 2001, Aachen (Germany).
23. F. Somenzi. CUDD: Colorado University Decision Diagram Package, Release 2.3.0. User's Manual and Programmer's Manual, 1998.