A Symbolic Approach to the Analysis of Multi-Formalism Markov Reward Models

Kai Lampka*, Markus Siegle**

*IT Department Uppsala University, Sweden
**Bundeswehr University Munich, Germany

Version of March 14, 2013

Abstract When modelling large systems, modularity is an important concept, as it aids modellers to master the complexity of their model. Moreover, employing different modelling formalism within the same modelling project has the potential to ease the description of various parts or aspects of the overall system. In the area of performability modelling, formalisms such as, for example, stochastic reward nets, stochastic process algebras, stochastic automata or stochastic UML state charts are often used, and several of these may be employed within one modelling project. This paper presents an approach for efficiently constructing a symbolic representation in the form of a Zero-suppressed Binary Decision Diagram (BDD) which represents the Markov Reward Model underlying a multi-formalism high-level model. In this approach, the interaction between the submodels may be established either by the sharing of state variables or by the synchronisation of common activities. It is shown that the Decision Diagram data structure and the associated algorithms enable highly efficient state space generation and different forms of analysis of the underlying Markov Reward Model, e.g. calculation of reward measures or asserting non-functional system properties by means of model checking techniques.

1 Introduction

Due to the complexity of today's systems, performance and dependability models should be built in a structured, i.e. modular and hierarchical fashion. Employing different modelling formalism within the same overall model can greatly assist the modeller in describing different aspects of the system in a clear and concise way. (Generalized) Stochastic Petri Nets, Stochastic Activity Neworks, Stochastic Automata, Stochastic Process Algebras, stochastic extensions of UML state charts and other modelling formalisms may thus be employed in an overall multi-formalism model. There are two basic forms of interaction between the submodels of an overall model, both well understood in the theory of concurrent processes: One of them is the sharing of state variables, which is a very general concept supported by literally every modelling formalism mentioned above. In

this approach, a subset of the state variables of a submodel is shared with one or more other submodels, so these state variables can be considered as global variables. The other form of interaction is synchronisation of common activities, which means that a designated subset of events may only take place jointly between two or more submodels. The consequence is that submodels have to wait for each other to perform these synchronised activities and are blocked as long as the partners are not ready to proceed.
For analysis, the high-level model description needs to be transformed into its low-level counterpart, of which this chapter assumes that it can be formalised as a Markov Reward Model (MRM), widespread in the performance / dependability literature. MRMs are continuous-time Markov chains augmented by reward / cost functions which enable the description and computation of a wide range of interesting performance and dependability measures. Examples for such measures are the expected accumulated reward gained during the mission of a spacecraft or the mean energy consumption per unit time of a production system.
A well-known drawback of state-based analysis is the problem of state space explosion, which means that the number of reachable states may grow exponentially in the number of concurrent activities of the high-level model. Among the techniques devised for coping with this problem, symbolic, i.e. decision diagram based approaches have shown to be particularly effective.

(Reduced Ordered Binary) Decision Diagrams (DD) are very useful for efficiently constructing and compactly representing the MRM underlying a high-level model description. Tools successfully employing DD-based techniques are stochastic model checkers like PRISM [31] CASPA [16] and SMART [35]. These tools are able to analyse models with hundreds of millions of states. However, when it comes to tools which support multiple modelling languages, e.g. the Möbius performance analysis framework [28], it is important that the DD-based analysis of MRMs is independent of the concrete model description method. This is not only because different entities of a system model might be described in a different method, but future extensions of the set of modelling formalisms should not require a re-implementation of the analysis engine. Independence of modelling formalism and DD-based analysis can be achieved by carrying out standard state space traversal and step-wise augmentation of the DD encoding the MRM. It is important, however, to note that these steps cannot practically be performed in a state-by-state manner, since this would lead to an unacceptable runtime overhead. Instead, operations which process sets of states and sets of transitions within one step are needed. Such operations can be provided by a DD environment, but they need to be used with great care and insight into the structure of the high-level model. Otherwise, negative effects[1] may destroy the efficiency of the approach.

In order to address these problems, this chapter introduces a scheme for efficiently constructing a DD-based representation of a high-level model's under-

[1]For example, the size of intermediate DD stuctures may increase dramatically during the incremental insertion of states / transitions, even if the final result is very compact.

lying MRM. The proposed technique does not depend on a specific modelling method and is therefore very well suited for multi-formalism models. This implies that the method needs to support different model composition schemes, where the paper discusses model composition via shared state variables and via synchronization of activities, both applicable within the same overall model. The state space generation method and the method for handling reward variables have been described before [20,21,18], but they are here placed for the first time in the context of a multi-formalism modelling environment.

Organisation of the chapter: Sec. 2 recapitulates basics of Markov Reward Modelling, introduces zero-suppressed Multi-terminal Binary Decision Diagrams (ZDDs) and shows how they can be employed for representing Markov Reward Models. This prepares the ground for the modelling formalisms independent scheme. Sec. 3 elaborates on the part which constructs the Continuous Time Markov Chain underlying a high-level model description. Sec. 4 presents the algorithms for constructing ZDD-based representation of reward functions and computing the performance metrics for the modelled system. Througout the paper, we employ a simple running example, specified as a multi-formalism MRM, to illustrate the concepts.

2 Background Material

Finite state Markov models constitute the common base for a wide range of different stochastic modelling formalisms. In the following, we briefly review the fundamentals of Markov Reward Models (MRM). This will be followed by a very brief introduction to high-level modelling techniques and composition schemes, i.e. methods for constructing high-level models in a hierarchic and compositional style. This will be followed by introducing zero-suppressed Multi-terminal Binary Decision Diagrams (ZDD) and showing how they can be used in a straightforward manner to represent Markov reward models. Overall, this provides the background for the discussion on how to derive symbolic model representations in the context of a multi-formalism high-level modelling environment efficiently.

2.1 Markov Reward Models

A (finite state) Markov Reward Model (MRM) consists of a Continuous Time Markov Chain (CTMC) and a set of reward functions defined for the states and the state-to-state transitions of the CTMC. In the following we detail on the relevant concepts.

Continuous Time Markov Chain (CTMC) A CTMC is a stochastic process $\{X(t) | \, t \in \mathbb{R}\}$ where $X(t)$ is interpreted as the state of the system at time t. In the context of this chapter, the state space is assumed to be a finite set of vectors ($\boldsymbol{s} \in \mathbb{S}$) of dimension n, where n is the number of state variables. Where appropriate, we will also employ indices like i and j in order to denote states. The distinctive property of Markov chains is the fact that they are memoryless,

which means that the future evolution depends only on the current state and not on the past history. The memoryless property implies that a Markov chain is not only independent of the sequence of visited states in the past, but also that the sojourn time T_i to be spent in the current state i is independent of the sojourn time already elapsed. Randomly distributed continuous sojourn times satisfying this property have exponential distribution: $Prob(T_i \leq t) = 1 - e^{-\lambda_i t}$. We formally define Continuous Time Markov chains (CTMC) as follows:

Definition 1. *A finite Continuous Time Markov chain (CTMC) is a triple* $C := (\mathbb{S}, T, \boldsymbol{\pi}(0))$ *where* $\mathbb{S}$ *is the finite set of system states.* T *is the matrix of transition rates among states, i.e. a mapping* $\mathbb{S} \times \mathbb{S} \mapsto \mathbb{R}_0^+$ *where* $\forall i \in \mathbb{S} : T(i,i) = 0$. *Vector* $\boldsymbol{\pi}(0)$ *defines an initial probability distribution on* $\mathbb{S}$.

In the context of this chapter, we are only concerned with time-homogeneous CTMCs, i.e. CTMCs where the transition rates are constant over time.

Reward Functions In addition to the CTMC, which captures the system behavior, reward functions constitute the other important part of a Markov reward model. Rate rewards depend on the system state of the CTMC, i.e. the value of the state variables, while impulse rewards are associated with the completion of transitions in the CTMC. A rate reward defines the reward gained per unit time by the model in a specific state. In contrast, an impulse reward defines the reward obtained by executing a specific activity in a specific state [32].
A set of rate and impulse reward functions, defined by the user on the high-level model, can be combined to form complex performance variables, i.e. the value of a performance variable p is assumed to be the sum of a set of rate or impulse reward functions. In the context of this chapter, the specific state- and/or transition-dependent reward values are assumed to be time-independent and we define them as follows:

Definition 2. *A rate reward* r *defined on a CTMC is a function* $\mathcal{R}^r : \mathbb{S} \to \mathbb{R}$.

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

Definition 3. *An impulse reward* a *of a CTMC is a function* $\mathcal{I}^a : \mathbb{S} \times \mathbb{S} \to \mathbb{R}$.

The set of all impulse rewards defined for a given CTMC is denoted $\mathcal{I}$. With these definitions we can now define:

Definition 4. *A Markov reward model (MRM) is a triple* $M := (C, \mathcal{I}, \mathcal{R})$, *where* C *is a CTMC,* $\mathcal{I}$ *is a set of impulse reward functions and* $\mathcal{R}$ *a set of rate reward functions.*

2.2 Numerical solution of (low-level) Markov Reward Models

The numerical solution of a Markov reward model involves the computation of state probability distributions and, on top of this, the computation of measures w. r. t. the reward functions.

Computing state probability distributions The state probability distribution of the Markov reward model (both for the transient and – if it exists – the stationary case) can be computed by standard numerical techniques for CTMC analysis. For the scope of this paper, we assume that the CTMC at hand is irreducible, as this simplifies the presentation. However, the methods presented are also applicable to the general case of Markov chains consisting of more than one bottom strongly connected components. Numerical methods for calculating steady-state or transient probabilities can be found in the literature, e.g., [37,8] and for their BDD-based variants [29,17,38,10,23,34].

Computing performability measures From the state probability distribution, measures related to the reward functions can be computed. Examples of such measures are the expected instant-of-time reward at steady state, or the mean number of transitions of a certain type (i.e. their throughput) per unit time. For simplicity we defined rate and impulse rewards as being state-/activity-dependent functions. In the following we briefly discuss the concept of rate and impulse rewards, details can be found, e. g. , in [33,32,6].

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

$$\mathcal{R}^r(i,t) = \pi_i(t) \cdot \mathcal{R}^r(i) \tag{1}$$

where $\pi_i(t)$ is the probability of being in state i at time point t and $\mathcal{R}^r(i)$ is the time-independent rate reward value of state i concerning rate reward r (cf. Def. 2). The probability $\pi_i(t)$ can be computed by standard numerical methods, namely the uniformisation algorithm for finite time points and iterative steady-state solvers for the limit $t \to \infty$. Since each state $i \in \mathbb{S}$ has its own rate reward value with respect to reward function r, one must simply sum the reward values over all states yielding the state-independent reward value $\mathcal{R}^r(t)$ at time-point t:

$$\mathcal{R}^r(t) := \sum_{i \in \mathbb{S}} \mathcal{R}^r(i,t) = \sum_{i \in \mathbb{S}} \pi_i(t) \cdot \mathcal{R}^r(i) \tag{2}$$

In addition to instant-of-time rewards, also interval-of-time and time averaged interval-of-time rewards are important. A rate reward obtained for a time interval $[t, t+\triangle t]$ can be computed as follows:

$$\mathcal{R}^r(t, t+\triangle t) := \sum_{i \in \mathbb{S}} \mathcal{R}^r(i) \cdot \bar{\pi}_i(t, t+\triangle t) \cdot \triangle t \tag{3}$$

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

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

By norming the computed value to the time period analyzed ($\frac{1}{\triangle t}$), the above interval-of-time reward measure can be converted into a time-averaged value. For the case $t \to \infty$ (steady-state), $\bar{\pi}_i(t, t+\triangle t)$ is simply replaced with the steady-state distribution π_i.

Note that due to the above definition of rate rewards, only those states contribute to $\mathcal{R}^r$ which have a state probability different from 0. We will come back to this issue in the next section, where SG reduction techniques are addressed.

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

$$\mathcal{I}^a(i,j,t,t+\triangle t) := \bar{\pi}_i(t,t+\triangle t) \cdot \triangle t \cdot \lambda_{i,j} \cdot \mathcal{I}^a(i,j) \tag{4}$$

where $\bar{\pi}_i(t,t+\triangle t)$ is defined as above and $\lambda_{i,j}$ is the transition rate between state i and j and $\mathcal{I}^a(i,j)$ is the impulse reward associated with this transition.

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

$$\mathcal{I}^a(i,t,t+\triangle t) := \sum_{j\in\mathbb{S}} \mathcal{I}^a(i,j,t,t+\triangle t) = \bar{\pi}_i(t,t+\triangle t) \cdot \triangle t \cdot \sum_{j\in\mathbb{S}} \mathcal{I}^a(i,j) \cdot \lambda_{i,j} \tag{5}$$

In order to obtain the "state-independent" impulse reward one simply needs to sum over all states, yielding:

$$\mathcal{I}^a(t,t+\triangle t) := \sum_{i\in\mathbb{S}} \mathcal{I}^a(i,t,t+\triangle t) = \sum_{i\in\mathbb{S}} \sum_{j\in\mathbb{S}} \mathcal{I}^a(i,j,t,t+\triangle t) \tag{6}$$

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

In the steady-state case, we restrict the discussion to time-averaged impulse rewards, so that $\bar{\pi}_i(t,t+\triangle t)$ in Eq. 4 can be replaced with the steady-state distribution π_i, where a subsequent norming to the length of the time interval of interest must follow. This yields the steady-state impulse reward rate from state i to state j:

$$\tilde{\mathcal{I}}^a(i,j) := \pi_i \cdot \mathcal{I}^a(i,j) \cdot \lambda_{i,j} \tag{7}$$

If this is employed in Eq. 5 and 6 one obtains:

$$\tilde{\mathcal{I}}^a(i) := \sum_{j\in\mathbb{S}} \tilde{\mathcal{I}}^a(i,j) = \pi_i \cdot \sum_{j\in\mathbb{S}} \mathcal{I}^a(i,j) \cdot \lambda_{i,j} \quad \text{and} \quad \tilde{\mathcal{I}}^a := \sum_{i\in\mathbb{S}} \sum_{j\in\mathbb{S}} \tilde{\mathcal{I}}^a(i,j) \tag{8}$$

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

2.3 High-level system modelling

In the following we will briefly introduce aspects of high-level system modelling. This introduction starts with the description of (sub-)models in a concise way, takes us through mechanisms for composing models in a hierarchical way and ends with the specification of performability measures.

High-level model description techniques High-level modelling is a key to state based system analysis, as it supports the compact, human readable system description, opposed to the error-prone specification of CTMCs with a huge number of states.
All high-level model specification methods discussed in the sequel have in common that a model M consists of a finite ordered set of discrete state variables ($\mathfrak{S}$) and a finite set of activities ($\mathcal{A}ct$). We use the term "activity" when referring to the high-level constructs (such as an action in a process algebra or an arrival in a queueing network) and the term "transition" when referring to the underlying Markov reward model. Thus, the execution of an activity in the high-level model is reflected by a state-to-state transition in the low-level model, i.e. in the CTMC.

Stochastic automata networks Stochastic automata networks [30,5,37] are a relatively low-level modeling approach, since the high-level model description resembles activity-labeled CTMCs. For compactly specifying complex systems, the modeler may combine sets of stochastic automata by activity synchronization (cf. Sec. 2.3). This leads to a network of stochastic automata which describes the behavior of a system in a compositional way. Since the individual stochastic automata do not contain any local variables, the state of a stochastic automata network is naturally described by a set of local state counters, each referring to the state of a specific stochastic automaton.

Stochastic state charts In recent years, state charts like the ones employed in UML have been extended to the Markovian case. In its simplest form, a stochastic state chart consists of a set of states, additional variables and transitions among these states, whose execution may modify the variables. Thus, a state of the state chart can be described by the current values of the local variables and an additional state counter, where we latter is employed for tracking the active state of the state charts. In order to specify timed behavior, activities of the state charts are labelled with rates, which specify an exponentially distributed execution delay. For modelling case distinctions, it is also possible to make use of a special "decision" node which is connected to one source state and possibly many successor states. The incoming edge of this node defines a Markovian activity, i.e. an activity which is executed after an exponentially distributed delay. The outgoing edges are equipped with probabilities, such that they allow the modelling of a probabilistic choice among the successor states.
By introducing the concept of initial and terminal states and referencing of (sub-) state charts, state charts can be organized in a modular fashion. It seems to

be straight-forward to also allow the composition of state charts via the sharing of variables and/or the joint execution of activities (cf. Sec. 2.3). A stochastic extension of UML state charts is described in [15].

Generalized stochastic Petri Nets A generalized stochastic Petri Net (GSPN) is a bi-partite directed graph, which consists of a set of activities (called "transitions" in the usual Petri net terminology) and a set of places. The current state of the system is given by the current marking, i.e. the number of tokens contained in each place, which means that each place of the GSPN can be regarded as a state variable of the overall model. The dynamic behaviour is specified by the activity firing rule. For specifying timed behaviour, activities are either executed after an exponential delay or instantaneously, where the race condition among competing activities must be resolved. A profound overview of GSPNs can be found in [3]. In order to enable the specification of complex GSPNs, it is possible to combine different (sub-)nets via the sharing of places. Furthermore, concepts known from stochastic automata and stochastic process algebras have been extended to the area of GSPNs, such that also the composition via activity-synchronization is applicable, e.g. [7,13] (cf. Sec. 2.3).

Stochastic Activity Networks Stochastic Activity Networks (SAN), introduced in [33], resemble GSPNs. A state of a SAN can also be described as a tuple of state variables, where each refers to a specific place of the net. In addition to GSPNs, SANs allow the use of so called input and output gates. These gates can be seen as an enrichment of the enabling predicates (guards) and the execution functions of the connected activities. SANs also allow the association of each activity with a set of "cases", where where the individual execution probability is determined by the specific case-individual weight. SANs allow the use of not only exponential distributions but also general distributions for the delay of activities. However, in the context of this chapter, we are occupied with Markovian models, thus we only consider Markovian and non-delayed activities. The SAN modelling formalism includes operators for composing submodels via the sharing of places (a general *Join* and a special *Replicate* operator). Furthermore, it is also possible to compose submodels via the joint execution of activities.

Stochastic process algebras A stochastic process algebra (SPA) specification is built with the help of operators for action prefix, (guarded) choice, (implicit or explicit) recursion, parallel composition, etc.. Consequently, the state of the process may be described by the values of the local process variables and a process counter. Actions can be timed, i.e. they are equipped with rates, or in some cases also non-delayed, i.e. instantaneous. Examples of stochastic process algebras can be found in [9,14,11,12].
An important concept of process algebras is constructivity: (a) Similar to stochastic automata, a system can be built in a compositional manner, where activity-synchronization (cf. Sec. 2.3) is employed for combining the individual process instances. (b) Process algebras are equipped with notions of equivalence of processes, which allows one to replace processes with simpler ones, such that the

overall system becomes smaller, but exhibits the same functional and timed behavior. In recent years, aspects of constructivity have been adopted to other high-level model description methods, where especially the compositional construction of high-level models plays an important role [13].

Remark: Timed and untimed activities. Most high-level Markov modelling methods feature the use of Markovian activities, i.e. activities whose execution delay is sampled from an exponential distribution, and the use of non-delayed, i.e. instantaneous activities. Therefore, after transformation of the high-level model into its underlying Markov reward model, two kinds of transitions between pair of states exist: immediate and timed ones. Timed transitions are taken with an exponential delay, whereas instantaneous transitions are taken immediately. It is is evident that within states with outgoing immediate and timed transitions, the latter will never be taken, known as the maximum progress assumption. The system will spend non-zero time only in states which can exclusively be left via outgoing timed transitions. States of such kind are denoted as tangible, whereas states to be left via immediate transitions are denoted as vanishing. In case a vanishing state can be left via more than one immediate activity, the non-determinism has to be resolved. This is done by assigning probabilities to each immediate transition. The result is a transition matrix T, where some entries refer to transition probabilities and some to transition rates. As known from the literature, e.g. [3,6], T can be converted into a pure transition rate matrix by eliminating all entries referring to vanishing states. This elimination can be done either at the level of the high-level model or at the level of the state space [2]. For simplicity we only consider high-level model descriptions with timed activities.

Composition of high-level model descriptions The high-level model is constructed in a modular and hierarchic way by specifying the type of interaction between a given set of submodels.

Sharing of state variables. If a high-level modelling formalism employs (local) variables, it is possible to compose submodels by merging sets of local variables ($\mathcal{J}$) [33]. This technique is commonly denoted as sharing or joining of state variables. In the following, we assume that (global) variables are shared among submodels. Local variables with the same name are consequently overloaded by the global definitions.

Joint activity execution. When composing submodels via joint activity execution, submodels are executed in parallel, but a subset of dedicated activities has to be executed jointly by all participating partners. Different approaches concerning the type of synchronizing activities exists, with different schemes for computing the rate of synchronised activities. In the following we will employ the operator $S_1 \|_{\mathcal{A}ct_{\mathcal{S}}} S_2$ for specifying the synchronization of submodel S_1 and S_2 over all activities appearing on the set $\mathcal{A}ct_{\mathcal{S}}$, which means that activities with the same label are executed synchronously.

Pure interleaving. If no interaction among the submodels takes place, one speaks of pure interleaving. In this case, the submodels are in fact (disjoint) partitions of the overall model executing concurrently. Pure interleaving is the special case of joint activity execution and sharing of state variables, in case the set of objects over which the submodels interact is empty. In this case we write $S_1 \| S_2$.

Running Example. Throughout this chapter, we employ a running example to illustrate the concepts. The example does not model a particular real system, it is just for demonstration purposes. The high-level description of the example is shown in Fig. 1. It is a multi-formalism model, where the overall model consists of two SPN submodels and three SPA submodels. Using SPA notation, the overall model is given by:

$$\begin{aligned} System := \; & processor(K, L) \| \\ & exceptionHandler(K) \|_{reset,\gamma}, exceptionHandler(L) \| \\ & User_1(K) \| User_2(L). \end{aligned}$$

The submodels interact via the global variables K and L, as they are passed by name into the instances of the submodels. The notation $\|_{reset,\gamma}$ means that the two instances of submodel $exceptionHandler(.)$ interact by synchronizing the execution of activity *reset*, with γ as the resulting transition rate of the synchronized activities. An example of a performance variable w. r. t. this high-level model is given in Fig. 6. There, we define a performance variable *Avail* which consists of the sum of two rate reward functions, r_1 and r_2 respectively (defined in a pseudo C notation). We obtain a rate reward value of 1 for each state where either K or L are 0 and a rate reward value of 2 for the initial system state (where both K and L are 0).

2.4 Symbolic represententation of Markov Reward Models

In this chapter we make use of **zero-suppressed** Multi-terminal Binary Decision Diagrams (ZDDs) [22] which we introduce in the following, together with their use for encoding MRMs.

The ZDD data structure and its associated 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 $f : \mathbb{B}^{|\mathcal{V}|} \to \mathbb{R}$, with $\mathcal{V} := \{\mathsf{v}_1, \ldots, \mathsf{v}_n\}$ as finite set of (Boolean) function variables. As we will see below, transition rates of the Markov reward model are interpreted as function values of transition functions and therefore stored within the terminal nodes of a ZDD.

A Binary Decision Tree (BDT) [25] is a binary tree $\mathsf{B} := \{\ \mathcal{V}, \mathcal{K}, \texttt{value}, \texttt{var}, \texttt{then}, \texttt{else}\ \}$, where:

- $\mathcal{V}$ is a finite and non-empty ordered set of boolean variables,

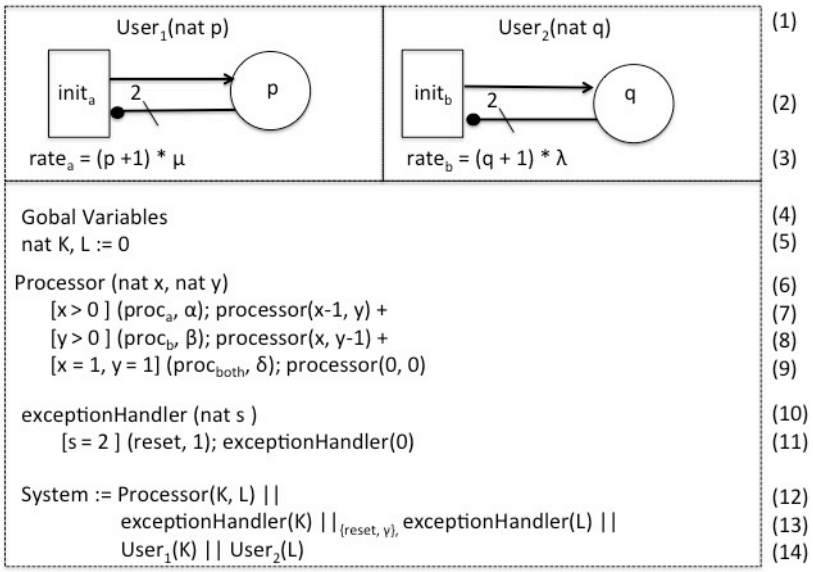


Figure 1. Running Example: A high-level model composed from heterogeneous sub-models by sharing variables and synchronizing activities.

- $\mathcal{K} = \mathcal{K}_T \cup \mathcal{K}_{NT}$ is a finite non-empty set of nodes, consisting of the disjoint sets of terminal nodes $\mathcal{K}_T$ and non-terminal nodes $\mathcal{K}_{NT}$,
- the mapping $\texttt{var} : \mathcal{K}_{NT} \mapsto \mathcal{V}$ is defined,
- the mapping $\texttt{value} : \mathcal{K}_T \mapsto \mathbb{B}$ is defined,
- the functions $\texttt{else}, \texttt{then} : \mathcal{K}_{NT} \mapsto \mathcal{K}$ are defined, and
- a function $\texttt{getRoot} : \mathsf{B} \mapsto \mathcal{K}$ yields the dedicated root node.

For the BDT to be ordered the following constraint must hold:

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

where $\succ \subseteq \mathcal{V} \times \mathcal{V}$ is a fixed ordering relation.
The BDT becomes a Multi-terminal BDT [1] if we allow the terminal nodes to also hold other values than 0 and 1, which can be easily achieved by extending the definition of function `value` accordingly. In the following are only concerned with the ordered multi-terminal case.

A Multi-terminal BDT becomes a Multi-terminal zero-suppressed BDD (ZDD for short) by applying the following two reduction rules:

(1) Non-terminal nodes whose 1-successor is the terminal 0-node are skipped.
(2) Isomorphic subgraphs are merged.

The first rule is the one originally proposed for zero-suppressed BDDs by Minato [27], and the second rule is the same as for standard BDDs.

For a ZDD it is important to know the set of Boolean variables on which it depends (since skipping a Boolean variable level means that that variable takes the value 0). This is especially important in multi-rooted DD environments as implemented in packages such as CUDD [36]. Here, ZDD-nodes lose their uniqueness if the represented functions have different sets of input variables. We define ZDDs to be partially shared if they do not necessarily have identical sets of Boolean variables, leading to different semantics of skipped levels while traversing the ZDDs. For this reason, we developed an algorithm for efficiently manipulating *partially shared* ZDDs, denoted as `ZApply`-algorithm, which is an extension of Bryant's famous `Apply`-algorithm [4]. Our algorithm which is described in [24] allows to apply boolean and arithmetic operators to ZDDs, where the operators mainly differ in the handling of the terminal nodes; we exploit the dualities $+ = \vee$ and $\cdot = \wedge$ in our notation from now on. Besides the `ZApply`-algorithm we also employ the algorithm $\texttt{Abstract}(\cdot, S, \mathsf{Z})$ which allows the all quantification of a ZDD Z w. r. t. variables of set S.

ZDD-based representation of Markov reward model By explicitly generating system states and state-to-state transitions, a CTMC for a given high-level model can be constructed. The CTMC defines a transition system $T \subseteq$

$(\mathbb{S} \times \mathcal{A}ct \times \mathbb{R}_+ \times \mathbb{S})$. Each transition of T can encoded by applying a binary encoding function `Encode`. This allows us to transforms each transition of the CTMC $i \xrightarrow{\lambda} j$ into a bit-vector. The individual bit positions correspond to the input variables of the ZDD, where we have the following convention:

- `s`-variables hold the encodings of a source state,
- `t`-variables hold the encodings of target states and
- the exponential (transition-)rate goes into the terminal node.

Following a widely used heuristics, the variables are ordered in an interleaved fashion: $\mathtt{s}_1 \prec \mathtt{t}_1 \prec \ldots \prec \mathtt{s}_n \prec \mathtt{t}_n$. This encoding yields a pseudo-Boolean function for each finite state CTMC.

In a similar fashion one may encoded the set of reachable states, the rate and impulse reward functions. The corresponding ZDDs take hereby solely the `s`-variables as input.

Running Example. For the high-level model specified in Fig. 1, Fig. 2.A - C sketches the above encoding scheme: The CTMC underlying the high-level model is depicted in Fig. 2.A. An example of a binary encoded (pseudo-) Boolean function and the corresponding ZDD-based representation is provided by Fig. 2.B and C. Dashed (solid) lines in the ZDDs indicate the value assignment 0 (1) to the corresponding Boolean variable on the respective path. The ZDDs are ordered, i.e. they have the same variable ordering one each path. As the order is from top to bottom, we omit the arrow heads on the node connecting edges. For clarity we also omitted the terminal 0-node and its incoming edges. The arrows pointing towards the top node of a ZDD signal the root node of the respective ZDD. The ZDDs are reduced and zero-suppressed, i.e. we do not show isomorphic nodes and we do not show nodes whose outgoing 1-edge leads to the terminal 0-node.
The variables k_i, k'_i, l_i, l'_i in the table of Fig. 2.B report the bit-value of the respective boolean (state) variable employed for encoding the transitions. The un-primed variables refer to the values of the bit positions of the encoded state variables before the transition has taken place, i.e. they refer to the `s`-variables in the above encoding scheme. The primed variables refer to the target state, i.e. the `t`-variables. Fig. 2.B and C also show the binary encodings and ZDD-based representation of a reward function w. r. t. the variables K and L of the high-level model, where we specified the reward function in Fig. 2.B and in a C-like expression.

3 Scheme for constructing a CTMC from its multi-formalism model description

The scheme presented in the following makes it possible to treat different high-level modelling formalisms in a more or less black-box manner. The scheme only builds on some basic properties derived from the structure of any high-level model making use of variables and activities.

3.1 Structural properties of high-level models

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

- $\mathfrak{S} := \{\mathfrak{s}_1, \ldots, \mathfrak{s}_n\}$ is an ordered set of state variables. As each state variable 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 state variables.
- $\mathcal{A}ct := \{n, l, \ldots, z\}$ 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 state variable $\mathfrak{s}_i$ and an activity l as connected *iff* $\mathfrak{s}_i$ influences the behavior of l or $\mathfrak{s}_i$ changes its value if activity l is executed.

Based on the connection relation $\mathcal{C}on$ we define the set of dependent state variables for each activity l:

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

The complementary set $\mathfrak{S}_l^I = \mathfrak{S} \setminus \mathfrak{S}_l^D$ denotes the independent state variables, i.e. the ones which neither influence the behaviour of activity l nor can they change their values once l is executed. Based on the above definition we define a projection function $\chi_l \colon \mathbb{N}_0^{|\mathfrak{S}|} \longrightarrow \mathbb{N}_0^{|\mathfrak{S}_l^D|}$ for each activity $l \in \mathcal{A}ct$. This function extracts the sub-vector w. r. t. activity l's set of dependent state variables 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. activity l). The above definition of dependent state variables 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:

$$(k, l) \in \mathcal{A}ct^{\mathcal{D}} \Leftrightarrow \mathfrak{S}_k^D \cap \mathfrak{S}_l^D \neq \emptyset. \tag{9}$$

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

$$\mathcal{A}ct_l^D := \{k \in \mathcal{A}ct \,|\, (l, k) \in \mathcal{A}ct^{\mathcal{D}}\}. \tag{10}$$

Please note that the above definition is reflexive, hence we have $l \in \mathcal{A}ct_l^D$.
The above sets are important for keeping explicit 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 model's underlying Markov Reward Model.

Running Example For the example of Fig. 1 we have $\mathcal{A}ct := \{init_a, init_b, proc_a, proc_b, proc_{both}, reset'_1, reset'_2\}$ as set of activities, where according to the above concept each activity possesses its individual sets of dependent and independent state variables, e. g. activity $init_a$ only contains variable K in its set of dependent state variable ($\mathfrak{S}_a^D := \{K\}$). As state variable K is shared among different submodels, we also have that $init_a$ is in a dependency relation with $proc_a$, $proc_{both}$ and $reset'_1$, which yields $\mathcal{A}ct_a^D := \{proc_a, proc_b, reset'_1\}$.

The operational semantics of a modelling formalism is irrelevant for the discussion to follow. This is because the proposed scheme relies on (partial) standard state space exploration, which makes it applicable for any kind of (state-based) high-level modelling formalism. The only two things which matter are the followings.

Guard functions For a given state $\boldsymbol{s}$ and a given activity l, a test method (aka guard) has to be available which decides whether or not l is executable in state $\boldsymbol{s}$. Formally: $guard_l : \mathbb{N}_0^{|\mathfrak{S}|} \to \{true, false\}$ where for $\mathsf{guard_l}(\boldsymbol{s}) = \mathsf{true}$ one says that activity l is enabled in state $\boldsymbol{s}$.

Transitions generating functions In case l is enabled in a (source) state $\boldsymbol{s}$, there is a method which returns the resulting (target) state $\boldsymbol{t} \in \mathbb{N}_0^{|\mathfrak{S}|}$. This function $\delta_l : \mathbb{N}_0^{|\mathfrak{S}|} \to \mathbb{N}_0^{|\mathfrak{S}|}$ is commonly denoted as transition function. In the following, target states will be equipped with superscripts which refer to the sequence of activities, i.e. δ_l executions, which led to the respective state, e. g. for $\boldsymbol{s}^{\,\omega}$ with $\omega := (\alpha, \ldots, \zeta) \in \mathcal{A}ct^*$ the state descriptor $\boldsymbol{s}^{\,\omega}$ refers to the activity execution sequence $\alpha, .., \zeta$ with $\boldsymbol{s}^{\,\omega} := \delta_\zeta(..\delta_\alpha(\boldsymbol{s}^{\,\epsilon})..)$. In this line, $\boldsymbol{s}^{\,\epsilon}$ addresses the high-level model's initial state.

Rate returning functions When executing l in a state $\boldsymbol{s}$ one needs a method which returns the execution rate of l. This function $\eta_l : \mathbb{N}_0^{|\mathfrak{S}|} \to \mathbb{R}_+$ is addressed as rate returning function. Commonly it is evaluate for the source state, i.e. a priori to the execution of the δ-function.

The concrete implementations of η_l, $\mathsf{guard_l}$ and δ_l are irrelevant for the discussion to follow. It is only required, that the evaluation of these functions solely depend on the dependent SVs of the respective activity.
Employing all η_l, $guard_l$ and δ_l-functions of a high-level model in a fixed point computation allows one to construct the activity-labeled CTMC for a given high-level model, the stochastic activity-labeled transition system $T \subseteq (\mathbb{S} \times \mathcal{A}ct \times \mathbb{R}_+ \times \mathbb{S})$ respectively. In this context, $\mathbb{S} \subseteq \mathbb{N}_0^{|\mathfrak{S}|}$ addresses the model's set of (reachable) states.

Running Example. For the example of Fig. 1 the guard function $\mathsf{guard_{proc_a}}$ is given by the following (C-like) expression $(K > 0)?true : false$. At model construction time, the local variable x of the instance of submodel *Processor* is

bound to global variable K, whereas local variable y is bound to global variable L (see Fig. 1). As another example one may consider activity $init_a$, with the expression $(K < 2)?true : false$ as guard function.
For $proc_a$ we also have δ_{proc_a} defined by the expression $K = K - 1$ and η_{proc_a} by the constant expression α. In case of activity $init_a$ we obtain the expressions $K = K + 1$ and $(K + 1) \cdot \mu$ for the δ, η-function respectively.

As already pointed out, we use joint variables and / or activity synchronisation for constructing models in a hierarchic manner. We assume that all state variables have a unique identifier throughout all submodels. Hence, sharing of state variables is realized by (re-)using the same (global) variable in the respective submodels. This naming convention is sufficient when it comes to the construction of the overall model's MRM at the level of decision diagrams.
Dealing with activity-synchronization also requires to define a naming convention. We assume that activities to be synchronized carry the same (main) label which we prime and index accordingly. The index refers to the respective instance of a submodel participating in the synchronization. Moreover, hierarchic use of synchronization operators, yield a multiset $\mathcal{A}ct_{\mathcal{S}}$ of activity labels, where each of its elements refers to a set of (sub)activities to be jointly executed, i.e. synchronized. However, sets of synchronizing (sub)activities referring to the same (main) label and which are directly adjacent within the model description need to be merged. E.g., $X\|_aY\|X\|_aY$ results in the multiset $\mathcal{A}ct_{\mathcal{S}} := \{a, a\}$ where each symbol references it set of (sub)activities, here : $\{a'_{X_1}, a'_{Y_1}\}$ and $\{a'_{X_2}, a'_{Y_2}\}$. In case of $X\|_aY\|_aX\|_aY$ we would need to merge the sets accordingly, yielding the (multi)set $\mathcal{A}ct_{\mathcal{S}} := \{a\}$ and the set $\{a'_{X_1}, a'_{Y_1}, \{a'_{X_2}, a'_{Y_2}\}$ of (sub)activities. In terms of the model of Fig. 1, we have a single element on the multiset $\mathcal{A}ct_{\mathcal{S}} := \{reset\}$ and a single set of subactivities $\{reset'_1, reset'_2\}$.
Beyond the naming convention and as far as the explicit exploration procedure is concerned, the handling of activities which take part in a synchronisation is straightforward: the (sub)activities are treated just like any other activity. Synchronization is only considered, when carrying out the ZDD-based computations for constructing a high-level model's CTMC.

3.2 The scheme at glance

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 ZDD-based computations, where we generate all possible transition interleavings by customized cross-product computations. In a nutshell, the proposed technique is round-based, where a round is made of (a) explicit state space exploration steps, (b) individual encoding of the generated state-to-state transitions, (c) pure symbolic, i.e. BDD-based manipulations of the generated transition system and (d) a re-initialization procedure for preparing the next round. Carrying out these steps in a fixed point computation, ultimately delivers the complete set of reach-

able states and transitions of the high-level model under analysis. These steps will be briefly sketched in the following paragraphs.

Explicit generation and encoding of transitions For generating symbolic representations of the state-to-state transition functions we employ the activity-local scheme as presented in [20]. This sheme gives one a ZDD for each activity which we denoted ZDD Z_l in case of a non-synchronizing activity l, and which we denoted ZDD $\mathsf{Z}_{l'_i}$ for a synchronizing activity l'_i.
It is important to note, that the number of executions of synchronizing activities may not be bounded when exploring them in isolation. This situation can be easily caught by simply limiting the number of explicit state explorations to be executed at once, i.e. per round.

Symbolic manipulations for obtaining set of states and transitions For obtaining a symbolic representation of the model's set of reachable states we employed a symbolic composition scheme which generates supersets of transitions. These potential transition functions are employed in a (standard) symbolic reachability analysis which at termination delivers a ZDD Z_R, which is the symbolic representation of a model's set of reachable states.

The algorithms implementing the above steps are shown in Fig. 3. In the following we detail now on selected aspects of the scheme.

3.3 Implementation details

For convenience we introduce the following sets:

$$\mathsf{D}_l := \{\mathbf{s}^{\,i}, \mathbf{t}^{\,i} | \mathfrak{s}_i \in \mathfrak{S}_l^D\} \text{ and } \mathsf{I}_l := \{\mathbf{s}^{\,i}, \mathbf{t}^{\,i} | \mathfrak{s}_i \in \mathfrak{S}_l^I\}, \tag{11}$$

where $\mathbf{s}^{\,i}$ and $\mathbf{t}^{\,i}$ refer to those Boolean variables which encode the value of dependent state variable $\mathfrak{s}_i$ 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.

In lines 1 - 3 of the top-level algorithm (Fig. 3.A) some data initialization is done: ZDD Z_R is set to the initial state $\boldsymbol{s}^{\,\epsilon}$ and the exploration and encoding buffers StateBuffer and TransBuffer are allocated. The buffer StateBuffer is used for holding tuples of states and activities, where the activities are supposed to be executed in the state. The buffer TransBuffer holds transitions to be explicitly encoded and inserted into a ZDD. Routine `Initialize`() fills StateBuffer 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, where we have a tuple for each activity enabled in the initial state.

Explicit Generation of states and transitions Routine `Explore()` generates a symbolic representation Z_l for each activity, including the (sub-)activities l'_i. The symbolic structure is generated by explicitly exploring and individually encoding the detected transitions. This step is repeated until no new transition can be detected or a pre-defined maximum on the explicit state exploration steps is reached. This latter maximum is necessary, as the exploration steps of any synchronizing (sub-)activity may not be finite when considered in isolation. The most distinguished feature of function `Explore` is the *selective* bfs exploration and encoding of transitions. A selective bfs 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 $\boldsymbol{s}_{d_k}^{\omega l}$ has not been tested for enabling activity k before ($k \in \mathcal{A}ct_l^D \wedge \boldsymbol{s}_{d_k}^{\omega l} \notin \mathsf{E}_k^i$. One may note that we do not need to expand all sequences of activity executions, as this is done on the level of symbolic reachability analysis.
For simplicity we store the activity markings which already have been tested on activity k in a respective symbolic structure denoted E_l^k.
The above ideas are implemented with the help of two complementary $while-loop$s of the algorithm of Fig. 3.C.

The upper loop fetches states and lists of activities $(\boldsymbol{s}^{\,l}, \mathcal{F}_{\boldsymbol{s}}^l)$ from the (exploration) buffer StateBuffer (line 3) and computes for each activity $k \in \mathcal{F}_{\boldsymbol{s}}^l$ the successor state $\boldsymbol{s}^{\,lk}$ and the transition rate w. r. t. the given source state $\boldsymbol{s}^{\,l}$ (line 5 and 6). The thereby established (stochastic) transition is inserted into the (encoding) buffer TransBuffer (line 7). and these steps are repeated until all activities of $\mathcal{F}_{\boldsymbol{s}}^l$ have been processed. This inner FOR-loop is repeated until all tuples of states and activities lists have been fetched from the exploration buffer StateBuffer.
Now, we execute the lower $while-loop$ which reads the individual transitions from the encoding buffer, individually encodes them and inserts the symbolically represented transition into the respective (activity-local) ZDD, where the encoding is implemented by function `Encode` (line 8 and 10). As long as the maximum number of exploration steps has not been reached (line 11), one computes the set of those activities which need to be considered for being explored in the state under consideration. The obtained set $\mathcal{F}_{\boldsymbol{s}^{\,l}}^l$ of activities is a subset of those activities which are in the dependency relation with the activity the execution of which brought the target state about (line 13), and their enabledness w. r. t. the state under consideration as not been tested in a previous round (line 14). For testing if an activity was already considered for execution we maintain a symbolic structure for each activity k (E_{lk}). This structure represents all activity-local markings the resp. activity was already tested or explored with. It is updated in line 16. The obtained set of enabled activities, together with the state under consideration is then inserted into the exploration buffer StateBuffer (line 19).
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 14). 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 implement a selective breadth-first search scheme.
Both while-loops of Algo. 3.C are executed sequentially until we reach a fixed point, i.e. $\mathsf{StateBuffer} = \emptyset \wedge FireCnt > MAX$ holds (line 22). Now, we have visited all states reachable from the initial state(s) through sequences of dependent activities. In case the maximum number of state enumerations has been reached, i.e. $FireCnt > MAX$ holds, we resume with state enumeration and transition encoding in the next round of partition-local explorations only, if re-initialization (`Initialize`) indicates the necessity of doing so. As already mentioned, this catches the case that possible exploration of (sub-)activities in isolation is unbounded a priori to their synchronization.
Once routine `Explore` has terminated that follows next, is the execution of routine `SymbReach` for obtaining a model's set of reachable states.

Symbolic reachability analysis `SymbReach`() (Fig. 3.D)) executes a symbolic reachability analysis in a fixed point iteration, organised here as a breadth first search; a more sophisticated scheme can be found in [20]. At first, the symbolic transition functions are extended by identity structures, assigned to those positions referring to the independent state variables of the respective activity. Building the union over all the extended symbolic transition functions yields the superset of transitions (line 2-9). Note that the synchronizing activities need to be combined via product-building before insertion of the identity structure, in order to implement their synchronized execution at the level of ZDDs. Once the symbolic transition function of the overall model is constructed (line 5), the actual symbolic reachability analysis can start.
Symbolic reachability analysis begins with the known states, i.e. either with the system's initial state or the states generated in previous rounds, initialization of Z_{unex} in line 1. In line 8 we compute the set of transitions emanating from the symbolically represented set of (currently) reachable states, whereas line 9 restricts these transitions to the encodings of target states and does a relabelling of the `t` into `s`-variables. The latter operation, denoted $\{\mathsf{t} \longleftarrow \mathsf{s}\}$, shifts the target states to source states. In fact, the above steps yield the set of newly reached states, which serve as input to the next iteration of the surrounding DO-WHILE-loop (line 6-7). Once a fixed point is reached, the set of (currently) reachable states has been generated.
The paper [19] introduces optimizations which make the symbolic composition procedure (line 2-5) a priori to symbolic reachability analysis unnecessary. At the bottom-line, [19] introduces a ZDD-operator `Execute` to be used in the DO-WHILE-loop of routine `SymbReach`(). This operates executes a symbolic image computation w. r. t. partial transition functions and for ZDD-based state representations. Moreover, instead of applying a pre-computed synchronisation-product of the synchronizing activities in a single step, they can simply be executed sequentially. For conciseness, we do not discuss this any further, the interested reader is referred to [19] for details.

At termination, routine `SymbReach`() has constructed the set of all currently reachable states. This may include states which result from the *interleaved execution of independent activities*. These states must be tested if they trigger new explicit model behaviour, not covered by the symbolic transition functions encoded by Z_T. For detecting such states we (re-)execute routine `Initialize`().

Re-initialization A re-initialization of the scheme is necessary, as the routine `Explore`() only extracts traces of dependent activities. Interleaving with independent activities is only done at the level of the symbolic representations. Therefore, states which are reached on execution traces consisting of the interleaved execution of independent activities may result in new model behavior, which requires a re-initializing of the scheme. This re-initialisation is realised by re-executing routine `Initialize`() which fills StateBuffer with the new elements, i.e. here with tuples consisting of a state and an activity. An activity l and a state $\boldsymbol{s}$ are considered by routine `Initialize` for exploration *iff* the activity-local marking of state $\boldsymbol{s}$ has not already been tested for enabledness by this activity (cf. line 2 of Algo. 3.B) and if the activity is enabled in this state (cf. line 6 of Algo. 3.B). If such states exist, a re-executing of the complete state space construction scheme must follow.
In case routine `Initialize`() as called in line 8 of the main routine does not find states triggering new transitions, a global fixed point has been reached and the scheme terminates.

Construction of the CTMC At termination, the above scheme delivers a set of (activity-local) transition systems, each induced by a dedicated activity and represented by a respective ZDD Z_l. Together with the ZDD-based representation of the set of reachable states Z_R, this allows us to construct the ZDD-based representation of the CTMC as follows:

$$\mathsf{Z}_R \cdot \left(\sum_{l \in \mathcal{A}ct_{\mathcal{S}}} \left(\prod_{\forall l'_i} \mathsf{Z}_{l'_i} \right) \cdot \rho_l \cdot \mathbb{1} <\mathsf{I}_l> \; + \sum_{k \in \mathcal{A}ct \setminus \mathcal{A}ct_{\mathcal{S}}} \mathsf{Z}_k \cdot \mathbb{1} <\mathsf{I}_l> \right) \qquad (12)$$

$\mathbb{1} <\mathsf{I}_l>$ is an identity structure over the set of activity l's set of independent SVs, their boolean counter parts respectively. The insertion of identity structures accounts for the fact that the independent variables maintain their values when the respective activity is executed. The rate ρ_l denotes the transition rate of the synchronised activities.
The above composition scheme resembles the Kronecker-operator-based approach of [30]. However, as the BDD-operators can cope with partition-wise nested variable orderings, the composition scheme of BDDs is much more flexible and can therefore be applied to almost arbitrarily structured high-level models which makes the here presented scheme extremely flexible.

Running Example . Fig. 4.A-F. illustrates the steps taken for the running example of Fig. 1. Fig. 4.A shows the ZDD encoding the set of reachable states at the beginning (when only the initial state (0,0) is known), and at the end of round 1. In Fig. 4.B, the fraction of the CTMC generated after the first round of the main routine is depicted. It contains only those transitions which can be reached on paths of dependent activities, i.e. the activities executed on the different paths have at least pair-wise, non-disjoint sets of variables. In part C of the figure, the transitions generated in the second round are shown, which are enabled due to the execution of independent activities. All of the respective (interleaved) execution sequences are generated on the level of ZDDs, rather than doing this explicitly.
The transitions resulting from the synchronisation of activities are also solely generated at the level of ZDDs. Non-synchronized executions are automatically discarded, due to the symbolic composition scheme. The remaining parts of the figure (D - F) depict the ZDDs encoding the activity-local transition functions, where part E shows where identity structures are inserted to reflect the fact that state variables not affected by a particular activity remain unchanged. The resulting transition system and set of reachable states has already been given in Fig. 2.C.

4 Scheme for the ZDD-based handling of performance variables

Performance variables, consisting of of rate reward and/or impulse reward definitions, enable the modeler to define complex performability measures on the basis of the high-level model, rather than on the level of the underlying CTMC.

Structural properties of rate reward returning functions Each rate reward function $\mathcal{R}^r(\boldsymbol{s}\ )$ has a set of input variables $\mathfrak{S}_r^D \subseteq \mathfrak{S}$ which is the set of state variables on which the computation of the rate reward value actually depends. Analogously to activities, we can extend this set to the Boolean variables used for encoding the respective state variables within the ZDD structures, denoted by the sets $\mathsf{D}_r^{\mathsf{s}}$ and $\mathsf{I}_r^{\mathsf{s}}$, (containing the rate reward dependent Boolean variables / the reward independent Boolean variables). Here we are only dealing with state encodings and not transition encodings, hence the above sets of Boolean variables are restricted to the set of s-variables. Analogously to the δ-functions, the concrete implementation of a reward returning function $\mathcal{R}_r$ is irrelevant.

Structural properties of impulse reward returning functions An impulse reward i is generated each time an activity k from the impulse reward's set of activities $\mathcal{A}ct_i$ is executed. The value of the impulse reward can be constant or state-dependent. This allows us to define the impulse reward returning function for impulse reward i as follows:

$$\mathcal{I}^i(\boldsymbol{s}\ ) := \sum_{k \in \mathcal{A}ct_i \cap \mathcal{A}_{\boldsymbol{s}}} \mathcal{I}_k^i(\boldsymbol{s}) \cdot \eta_k(\boldsymbol{s})$$

where $\mathcal{A}_{\boldsymbol{s}}$ is the set of activities enabled in state $\boldsymbol{s}$, and η is the rate returning function, both introduced earlier. $\mathcal{I}_k^i$ is the impulse reward returning function of impulse reward i and w. r. t. activity k. This allows for greater flexibility, as an impulse reward can be associated with different activities. Moreover, an activity may produce different reward values for different impulse reward definitions. In the following we assume that the computation of the impulse reward returning function $\mathcal{I}_k^i$ solely depends on those positions of $\boldsymbol{s}$ which actually correspond to state variables of $\mathfrak{S}_k^D$, otherwise $\mathfrak{S}_k^D$ needs to be adapted accordingly. An adjustment is irrelevant for the scheme for generating the state space, as the generation of reward values for transitions and states only take place once state space construction has terminated. Analogously to rate rewards, we derive the sets $\mathsf{D}_k^{\mathbf{s}}$ and $\mathsf{I}_k^{\mathbf{s}}$ which contain the dependent and independent variables.

For computing the mean and variance of the user-defined performance variables, the top-level routine ComputePV() defined in Fig. 5.A exploits the following algorithms.

(1) Algorithm ComputeStateProbabilities for computing the state probability distribution. This algorithm is not explained here. An overview of numerical solution methods can be found e.g. in [37]. The adaptations of the numerical solution methods to the case of BDD-based matrix representations will be briefly sketched below.
(2) Algorithms MakeRateRewards and MakeImpulseRewards which generate the ZDD-based representations of the user-defined rate and impulse reward functions.
(3) Algorithm ComputeRew which combines reward information with the computed state probabilities.

In the following we will explain these algorithms.

4.1 Computing state probabilities

Function ComputeStateProbabilities (line 3 of algorithm of Fig. 5.A) delivers the vector of state probabilities. The iterative solvers follow an approach in which the generator matrix is represented by a symbolic data structure and the probability vectors are stored as arrays. For details please refer to [29,38,23].
If n Boolean variables are used for state encoding, there are 2^n potential states, of which only a small fraction may be reachable. Allocating entries for unreachable states in the vectors would waste memory space, thereby severely restricting the applicability of the algorithms. Therefore a dense enumeration scheme for the reachable states is implemented via the concept of offset-labeling, as first suggested in [29] for the MTBDD data structure. While traversing the MTBDD representation of a matrix, in order to extract a matrix entry, the row and column index in the dense enumeration scheme can be determined from the offset values, basically by adding the offsets of those nodes where the `then`-Edge is taken. In other words, the offsets are used to map the **s** and **t** vectors to a pair

(r, c) of dense row and column indices. Using ZDDs we adapted the concept of offset-labeling:

- With standard Multi-terminal Binary Decision Diagrams (MTBDDs), skipped nodes (corresponding to don't cares) must be reinserted, because they carry an offset (which is relevant if their `then`-edge is followed). With ZDDs, skipped nodes correspond to zero-valued variables for which the offset is irrelevant. Therefore, in the ZDD case, skipped nodes do not have to be reinserted, which keeps the symbolic data structure compact.
- Similar to the MTBDD case, a ZDD node may have to be duplicated if the offset of a shared node is different on different paths (also called "offset clash").

The space efficiency of ZDD-based matrix representation comes at the cost of computational overhead, caused by the recursive traversal of the DD during access to the matrix entries. Analogously to [29], we replace the lower levels of the ZDDs by explicit sparse matrix representations, which works particularly well for block-structured matrices. We call the resulting data structure *hybrid offset-labeled* ZDD (HO ZDD). The level at which one replaces the remaining ZDD-levels with a sparse matrix representation is called *sparse level*. It depends on the available memory space, i.e. there is a typical time/space tradeoff.
For numerical analysis, the Gauss-Seidel (GS) method and its over-relaxed variant typically exhibit much better convergence than the power method, Jacobi (JAC) or Jacobi-Over-relaxation (JOR). However, Gauss-Seidel requires row-wise access to the matrix entries, which, unfortunately, cannot be realized efficiently with DD-based matrix representations. As a compromise we adapt the so-called pseudo-Gauss-Seidel (PGS) iteration scheme [29] to the case of HO ZDDs. For doing so the overall matrix is partitioned into blocks (not necessarily of equal size, due to unreachable states). Within each block, access to matrix entries is in arbitrary order, but the blocks are accessed in ascending order. PGS requires only one complete iteration vector and an additional vector whose size is determined by the maximal block size. Given a HO ZDD which represents the matrix, each inner node at a specific level corresponds to a block. Pointers to these nodes can be stored in a sparse matrix, which means that effectively the top levels of the HO ZDD have been replaced by a sparse matrix of block pointers. The level at which the root nodes of the matrix blocks reside is called *block level*. Overall, this yields a memory structure in which some levels from the top and some levels from the bottom of the HO ZDD have been replaced by sparse matrix structures. The choice of adequate sparse and block levels for converting the ZDD into sparse matrix structures is an optimization problem. In general, increasing the number of top ZDD levels improves convergence of the PGS scheme, and replacing more levels at the bottom of the ZDD, i.e. turning the terminal nodes into sparse matrix structures, improves speed of access. Since ZDDs are often more compact, their processing requires less CPU-time, if compared to MTBDDs. Due to their lower memory requirements they furthermore allow the removal of more levels, resulting in an additional speed-up. If the *block-*

level meets the *sparse-level*, as has been described in [26] and [38], all DD levels have disappeared and the PGS scheme becomes a proper GS scheme, but in most interesting cases this situation cannot be realized since memory is at a premium. Our experiments, carried out in [38], showed that using ZDDs an optimal choice for the block-levels to be removed often lies beyond half of the DD-levels. For comparison, the heuristic developed in [29] for MTBDDs suggested a third.

4.2 Generating ZDD-based representations of rate rewards

Once the state probabilities are computed, we call the functions MakeRateRewards and MakeImpulseRewards for computing symbolic representations of the rate reward and impulse reward functions associated with the user-defined performance variables. Again, our algorithms exploit locality, such that the explicit evaluation of reward functions is limited to a small fraction of states, rather than evaluating a reward function for each state.

Algorithm MakeRateRewards as specified in Fig. 5.B consists of two nested loops. The outer FOR-loop processes each rate reward definition contained in a user-defined performance variable, whereas in the inner WHILE-loop sets of states are processed. Z_U, initialized with the set of reachable states in line 2, contains all those states which still need to be considered for reward computation. First, an arbitrary state is extracted from the set of reachable states (line 4). This state is reduced to the positions referring to the rate reward-dependent state variables by an abstraction operation. Next, r's rate reward is calculated w. r. t. the extracted state vector, which can be done by executing the respective rate reward function $\mathcal{R}_r(\boldsymbol{s})$ (line 6). In case the obtained reward rew is not equal to zero, Z_s, Z_R and rew are multiplied. The newly obtained pairs of full (!) states and rate rewards are then added to the previously computed pairs as represented by ZDD R^r (line 8). Note that the operation $\mathsf{Z}_s \cdot \mathsf{Z}_U$ in line 8 yields the set of reachable states which are all equivalent concerning the variables of $\mathsf{D}^{\mathsf{s}}_r$. Line 9 removes all these states from the set of states represented by Z_U. Once all rate reward-dependent partitions of Z_R are processed, i.e. once Z_U is empty, the reward computation proceeds with the next rate reward (outer FOR-loop). At termination, a ZDD-based representation for each rate reward function is generated.

4.3 Generating ZDD-based representations of impulse rewards

Fig. 5.C. shows the algorithm MakeImpulseRewards for calculating impulse reward functions. Each activity may generate different impulse rewards for different impulse reward definitions, thus the algorithm iterates over three nested loops. The two outer `for`-loops process each impulse reward definition and its respective sets of activities. The inner `while`-loop processes one state for each activity-local marking in which the activity is enabled and calculates the respective impulse reward (line 5-12). In case the obtained impulse reward for a state is not equal to zero, the ZDD-based representation of all equivalent states (i.e. those with the same activity-local marking) is multiplied with the impulse

reward *imp* (line 10-11). Due to the construction of Z_U (line 4), the obtained pairs of states and impulse rewards are automatically weighed by the execution rate of the activity. The newly obtained pairs of full states and weighed impulse rewards are then added to the set of previously computed impulse rewards. This procedure is repeated until all "*activity-local*" markings are processed, i.e. until ZDD Z_U is empty.

4.4 Computing the performability measures

From the symbolic representations $\mathsf{Z}_{rate}, \mathsf{Z}_{imp}$ and the probability vector, the first and second moment of performance variable p is compututed by simultaneously traversing the offset-labeled ZDD Z_R^o and Z_{rate}, Z_{imp} respectively. This is the idea behind algorithm ComputeRew of Fig. 5.D: while traversing the ZDDs, the state index of the traversed path is obtained by summing over the offsets of nodes (line 7-8 of algorithm of Fig.5.D). Once a terminal non-zero node for Z_R^o is reached, the index of the state currently under consideration is determined, here contained in variable off (offset). The index allows one to fetch the respective probability value from the vector of state probabilities and compute (successively) the mean and second moment of the user-defined reward (line 2-3).

4.5 The top-level algorithm

Routine ComputePV() puts everything together: in line 1, we restrict the CTMC to its reachable portion. This is followed by applying the offset-labeling scheme, thereby generating ZDD Z_R^o. Depending on the employed solution method, the state probability vector refers either to the steady state probability distribution or the transient state probabilities at time t. In line 4 and 5 our algorithm generates the symbolic representation for each rate and impulse reward function as contained in the user-defined performance variables.
The FOR-loop of lines 6-15 processes each of the user-defined performance variables, where in lines 7-8 the respective reward functions are aggregated and where the call to ComputeRew delivers the mean and second moment of the reward function under consideration. After using these values for computing the variances of the rate and impulse reward (lines 13-14) the algorithm resumes with the next performance variable until all user-defined performance variable have been processed.

4.6 Running Example

Continuing our running example, the ZDD-based representation of the user-defined performance variable *Avail* (already introduced earlier) is depicted in Fig. 6. It consists of the sum of two rate reward functions, r_1 and r_2 respectively (see Example in Sec. 2.3). Fig. 6.B depicts the ZDDs obtained for the rate reward functions R_{r_1}, the skipping of the variables k_1, k_2 refers to the fact that they

are not in the support of rate reward function R_{r_1}. The combined rate reward, i.e. a ZDD-based representation of performance variable *Avail*, is also given Fig. 6.B, together with the offset labeled ZDD Z_{reach}. As the latter encodes the set of reachable states, it allows a dense numbering of the state space via the concept of offset-labelling as already mentioned above. Fig. 6.C shows parts of the recurison tree of algorithm ComputeRew when executed on ZDDs Z_{Avail} and Z_{reach}. The return values are indicated at the bottom-line. For conciseness, the parameters m and v are omitted, as they are only used for propagating the return values (mean and second moment of the reward function).

5 Conclusion

This chapter reviewed an efficient semi-symbolic technique for constructing a compact, ZDD-based representation of a high-level model's reachable state space, as well as its underlying Markov Reward Model. As its key feature, the presented approach is independent of the modelling formalism, which makes it applicable in the context of multi-formalism modelling environments (as shown by a small running example). Our implementations were carried out in the Moebius performance analysis framework [28], but the method could easily be adapted to a wide range of tools. The independence of the modelling modelling formalism has its price, namely it comes with explicit enumeration and encoding of states and transitions, and reward evaluations for individual states. For keeping this overhead as low as possible, the presented technique exploits the dependency relation among activities, reward returning functions and state variables of the high-level model. This features a selective handling of individual states, thereby effectively limiting CPU-time and peak memory consumption. Without such a feature, any BDD-based technique handling individual states could not succeed, since the peak number of BDD-nodes and the related memory requirements easily exceed the capacity of today's computers.

References

1. *Formal Methods in System Design: Special Issue on Multi-terminal Binary Decision Diagrams*, Volume 10, No. 2-3, 1997.
2. J. Bachmann, M. Riedl, J. Schuster, and M. Siegle. An Efficient Symbolic Elimination Algorithm for the Stochastic Process Algebra Tool CASPA. In *35th Int. Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM'09)*, pages 485–496. Springer LNCS 5404, 2009.
3. G. Balbo, G. Conte, S. Donatelli, G. Franceschinis, M. Ajmone Marsan, and M. Ajmone Marsan. *Modelling with Generalized Stochastic Petri Nets.* JOHN WILEY & SONS, 1995.
4. R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
5. P. Buchholz. *Die strukturierte Analyse Markovscher Modelle.* PhD thesis, Universität Dortmund, Dortmund (Germany), 1991.

6. G. Ciardo, A. Blakemore, P. F. J. Chimento, J. K. Muppala, and K. S. Trivedi. Automated generation and analysis of Markov reward models using Stochastic Reward Nets. *IMA Volumes in Mathematics and its Applications*, 48:145–191, 1993.
7. 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.
8. Bennett L. Fox and Peter W. Glynn. Computing poisson probabilities. *Commun. ACM*, 31(4):440–445, 1988.
9. N. Götz. *Stochastische Prozessalgebren – Integration von funktionalem Entwurf und Leistungsbewertung verteilter Systeme.* PhD thesis, Friedrich-Alexander-Universität Erlangen–Nürnberg, Erlangen (Germany), 1994.
10. S. Harwarth. Computation of transient state probabilities and implementing Möbius' "state-level abstract functional interface" for the data structure ZDD, 2006. Master Thesis. University of the Federal Armed Forces Munich (Germany).
11. H. Hermanns, U. Herzog, U. Klehmet, V. Mertsiotakis, and M. Siegle. Compositional performance modelling with the TIPPtool. In R. Puigjaner, N. Savino, and B. Serra, editors, *10'th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'98)*, LNCS 1469, pages 51–62. Springer Verkag, 1998.
12. H. Hermanns, U. Herzog, and V. Mertsiotakis. Stochastic Process Algebras - Between LOTOS and Markov Chains. *Computer Networks and ISDN Systems, 30 (9-10)*, pages 901–924, 1998.
13. Holger Hermanns, Ulrich Herzog, Vassilis Mertsiotakis, and Michael Rettelbach. Exploiting stochastic process algebra achievements for generalized stochastic petri nets. In *PNPM '97: Proc. of the 6th International Workshop on Petri Nets and Performance Models*, page 183, Washington, DC, USA, 1997. IEEE Computer Society.
14. J. Hillston. *A Compositional Approach to Performance Modelling.* PhD thesis, University of Edinburgh, Edinburgh (UK), 1994.
15. D. Jansen. *Extensions of Statecharts: with probability, time, and stochastic timing.* PhD thesis, University of Twente, Enschede (The Netherlands), 2003.
16. 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.
17. M. Kwiatkowska, R. Mehmood, G. Norman, and D. Parker. A symbolic out-of-core solution method for Markov models. *Electr. Notes Theor. Comput. Sci.*, 68(4):589–604, 2002.
18. 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.
19. K. Lampka. A new algorithm for partitioned symbolic reachability analysis. *ENTCS 223*, Workshop on Reachability Problems, 2008.
20. K. Lampka and M. Siegle. Activity-Local State Graph Generation for High-Level Stochastic Models. In *MMB'06*, pages 245–264, 2006.
21. K. Lampka and M. Siegle. Analysis of Markov Reward Models using Zero-supressed Multi-terminal decision diagramms. In *Proc. of VALUETOOLS 2006 (CD-edition)*, 2006.
22. 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.

23. Kai Lampka, Stefan Harwarth, and Markus Siegle. Can matrix-layout-independent numerical solvers be efficient? In *Proceedings of the International Workshop on Tools for solving Structured Markov Chains 2007*, volume 2, pages 1–9, Nantes, Oct 2007. ACM CD edition.
24. Kai Lampka, Markus Siegle, Joern Ossowski, and Christel Baier. Partially-shared zero-suppressed multi-terminal bdds: concept, algorithms and applications. *Form Methods System Design*, 36:198–222, Jun 2010.
25. C.Y. Lee. Representation of Switching Circuits by Binary-Decision Programs. *Bell Systems Technical Journal*, 38:985–999, 1959.
26. R. Mehmood. *Disk-based techniques for efficient solution of large Markov chains.* PhD thesis, University of Birmingham, University of Birmingham (U.K.), 2004.
27. 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.
28. Möbius page. www.mobius.uiuc.edu.
29. D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems.* PhD thesis, University of Birmingham, Birmingham (U.K.), 2002.
30. 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.
31. PRISM web page. www.prismmodelchecker.org.
32. W. H. Sanders and J. F. Meyer. A unified approach for specifying measures of performance, dependability, and performability. *Dependable Computing and Fault-Tolerant Systems: Dependable Computing for Critical Applications*, 4:215–237, 1991.
33. W.H. Sanders. *Construction and solution of performability models based on stochastic activity networks.* PhD thesis, University of Michigan, 1988.
34. J. Schuster and M. Siegle. A symbolic multilevel method with sparse submatrix representation for memory-speed-tradeoff. In *14th GI/ITG Conference on Measurement, Modeling and Evaluation of Computer and Communication Systems*, 2008.
35. SMART web page. www.cs.ucr.edu/∼ciardo/SMART.
36. F. Somenzi. CUDD: Colorado University Decision Diagram Package, Release 2.3.0. User's Manual and Programmer's Manual, 1998.
37. W. J. Stewart. *An Introduction to the solution of Markov Chains.* Princeton University Press, Princeton, NJ (USA), 1994.
38. D. Zimmermann. Implementierung von Verfahren zur Lösung dünn besetzter linearer Gleichungssysteme auf Basis von Zero-suppressed Multi-terminalen Binären Entscheidungsdiagramme, 2005. Diplomarbeit angefertigt an der Universität der Bundeswehr Neubiberg (Germany) (Master Thesis).

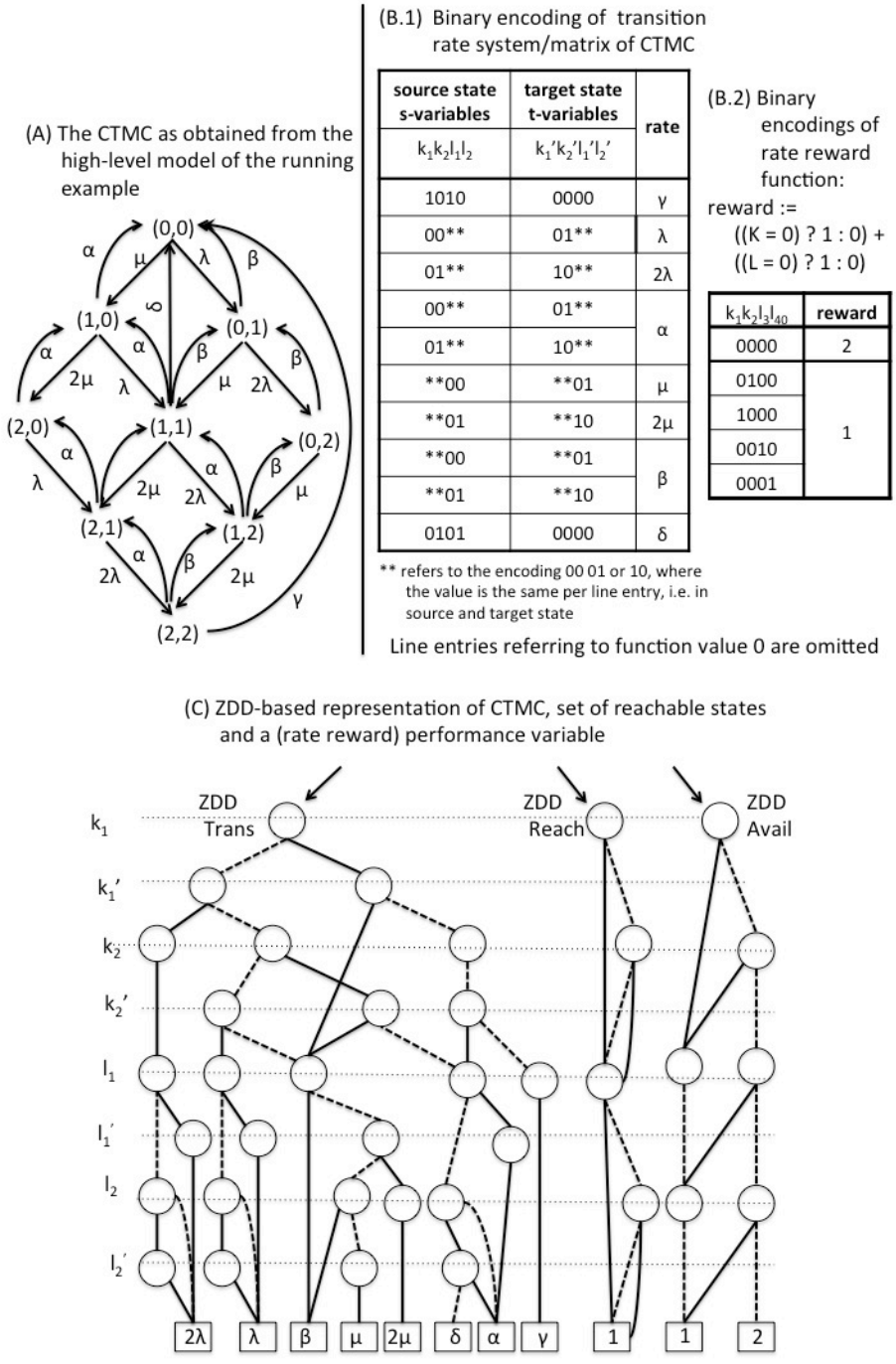
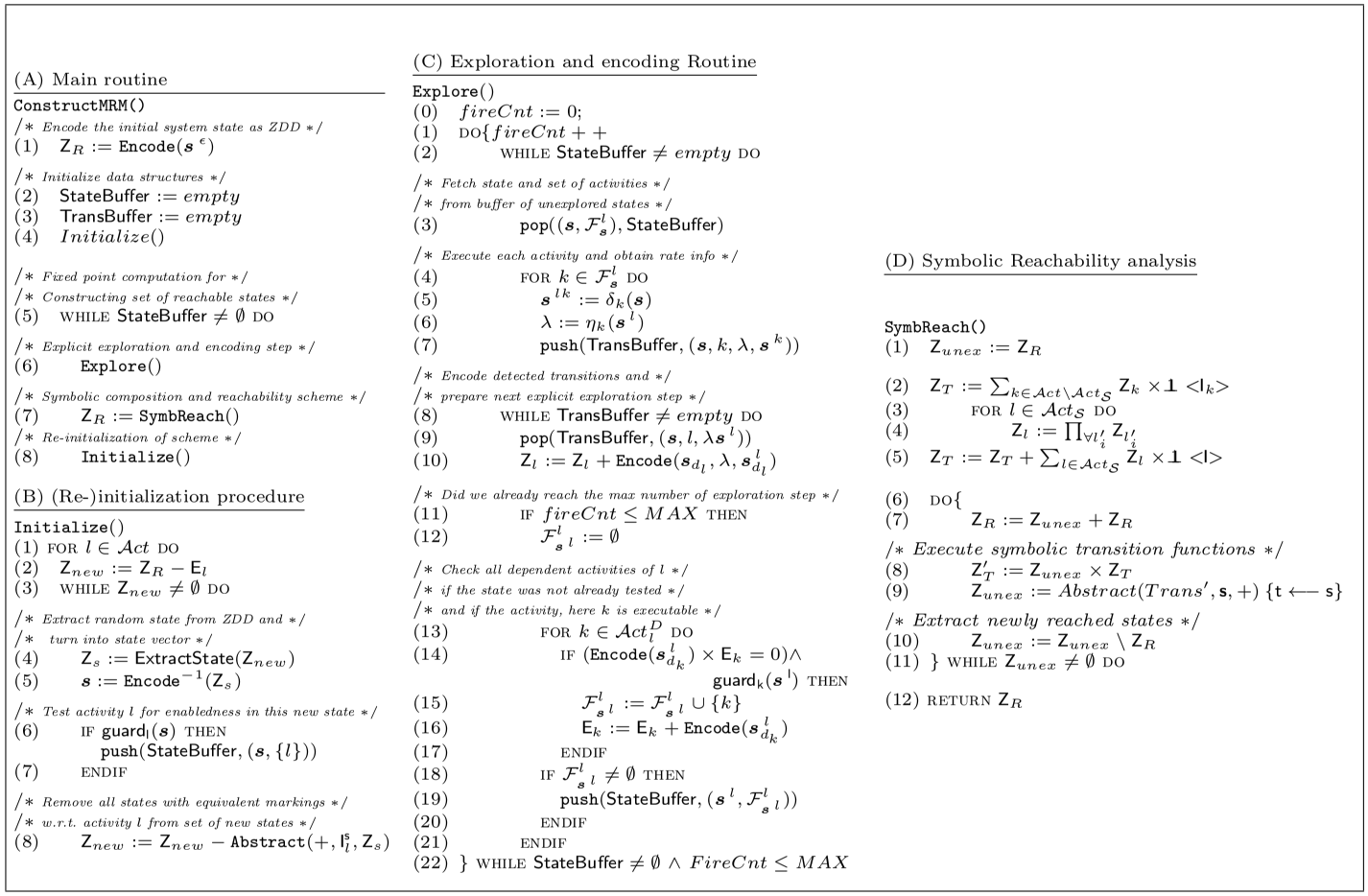


Figure 2. Running example: CTMC and ZDD-based representations of the underlying Markov reward model



(A) Main routine

```
ConstructMRM()
/* Encode the initial system state as ZDD */
(1)  Z_R := Encode(s^ε)

/* Initialize data structures */
(2)  StateBuffer := empty
(3)  TransBuffer := empty
(4)  Initialize()

/* Fixed point computation for */
/* Constructing set of reachable states */
(5)  WHILE StateBuffer ≠ ∅ DO

/* Explicit exploration and encoding step */
(6)      Explore()

/* Symbolic composition and reachability scheme */
(7)      Z_R := SymbReach()
/* Re-initialization of scheme */
(8)      Initialize()
```

(B) (Re-)initialization procedure

```
Initialize()
(1) FOR l ∈ Act DO
(2)   Z_new := Z_R − E_l
(3)   WHILE Z_new ≠ ∅ DO

/* Extract random state from ZDD and */
/*  turn into state vector */
(4)      Z_s := ExtractState(Z_new)
(5)      s := Encode^{-1}(Z_s)

/* Test activity l for enabledness in this new state */
(6)      IF guard_l(s) THEN
           push(StateBuffer, (s, {l}))
(7)      ENDIF

/* Remove all states with equivalent markings */
/* w.r.t. activity l from set of new states */
(8)      Z_new := Z_new − Abstract(+, I^s_l, Z_s)
```

(C) Exploration and encoding Routine

```
Explore()
(0)   fireCnt := 0;
(1)   DO{fireCnt++
(2)       WHILE StateBuffer ≠ empty DO

/* Fetch state and set of activities */
/* from buffer of unexplored states */
(3)           pop((s, F^l_s), StateBuffer)

/* Execute each activity and obtain rate info */
(4)           FOR k ∈ F^l_s DO
(5)             s^{lk} := δ_k(s)
(6)             λ := η_k(s^l)
(7)             push(TransBuffer, (s, k, λ, s^k))

/* Encode detected transitions and */
/* prepare next explicit exploration step */
(8)           WHILE TransBuffer ≠ empty DO
(9)             pop(TransBuffer, (s, l, λs^l))
(10)            Z_l := Z_l + Encode(s_{d_l}, λ, s^l_{d_l})

/* Did we already reach the max number of exploration step */
(11)          IF fireCnt ≤ MAX THEN
(12)            F^l_{s^l} := ∅

/* Check all dependent activities of l */
/* if the state was not already tested */
/* and if the activity, here k is executable */
(13)            FOR k ∈ Act^D_l DO
(14)              IF (Encode(s^l_{d_k}) × E_k = 0)∧
                                  guard_k(s^l) THEN
(15)                F^l_{s^l} := F^l_{s^l} ∪ {k}
(16)                E_k := E_k + Encode(s^l_{d_k})
(17)              ENDIF
(18)            IF F^l_{s^l} ≠ ∅ THEN
(19)              push(StateBuffer, (s^l, F^l_{s^l}))
(20)            ENDIF
(21)          ENDIF
(22)  } WHILE StateBuffer ≠ ∅ ∧ FireCnt ≤ MAX
```

(D) Symbolic Reachability analysis

```
SymbReach()
(1)  Z_unex := Z_R

(2)  Z_T := Σ_{k∈Act\Act_S} Z_k ×1 <I_k>
(3)      FOR l ∈ Act_S DO
(4)          Z_l := Π_{∀l'_i} Z_{l'_i}
(5)  Z_T := Z_T + Σ_{l∈Act_S} Z_l ×1 <I>

(6)  DO{
(7)      Z_R := Z_unex + Z_R
/* Execute symbolic transition functions */
(8)      Z'_T := Z_unex × Z_T
(9)      Z_unex := Abstract(Trans', s, +) {t ⟵ s}
/* Extract newly reached states */
(10)     Z_unex := Z_unex \ Z_R
(11) } WHILE Z_unex ≠ ∅ DO

(12) RETURN Z_R
```

Figure 3. Algorithms for the exploration scheme

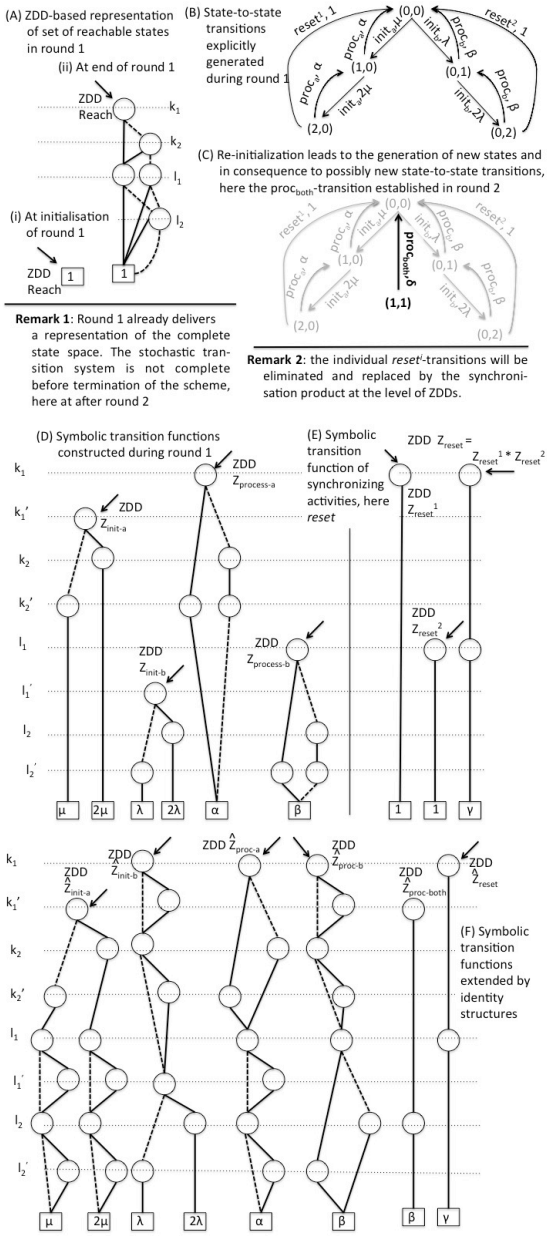


Figure 4. Running Example: From a multi-formalism model to the ZDD-based representation of the activity-local transition functions

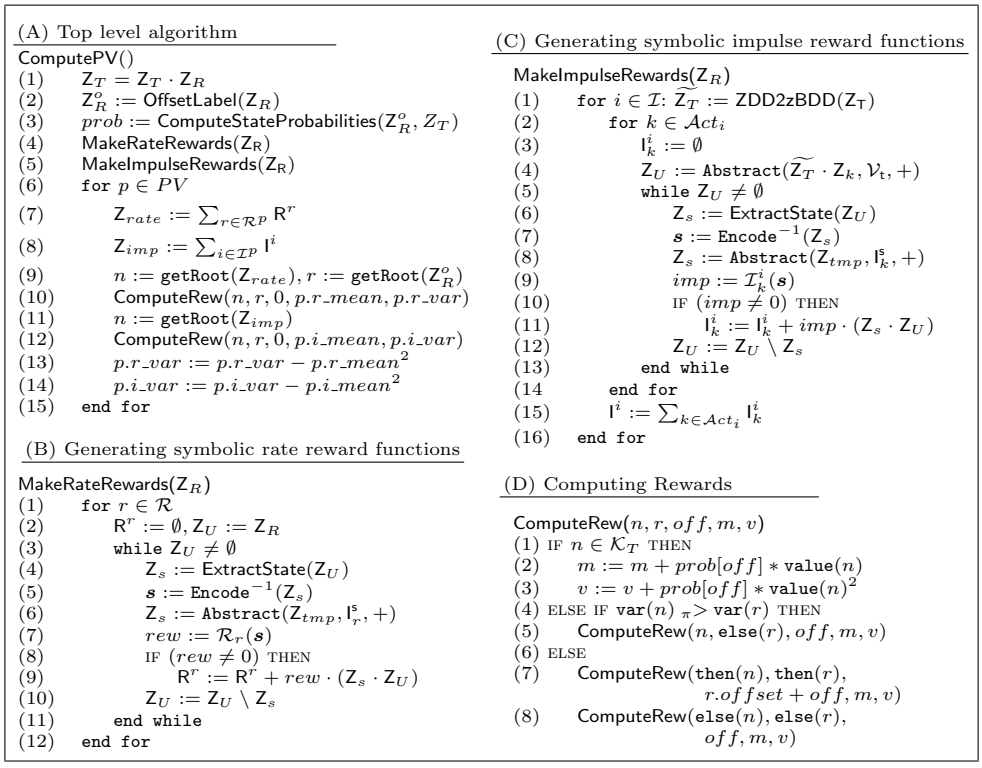


Figure 5. Algorithms for the handling performance variables

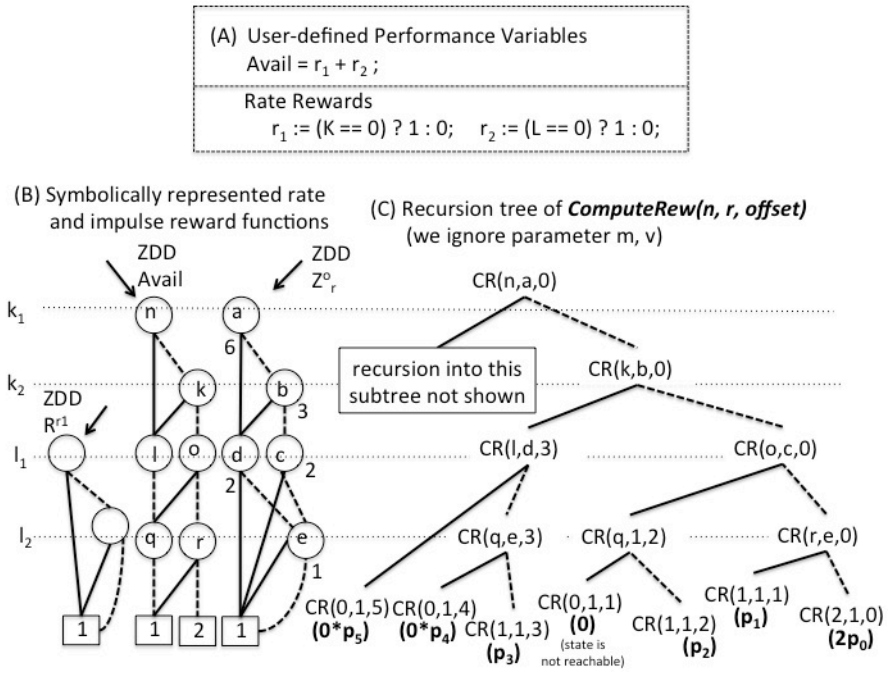


Figure 6. ZDD-based representation of Performance Variables