SPDL Model Checking via Property-Driven State Space Generation

Matthias Kuntz, Boudewijn R. Haverkort

University of Twente,
Faculty for Electrical Engineering, Mathematics and Computer Science

Abstract. In this report we describe how both, memory and time requirements for stochastic model checking of SPDL (stochastic propositional dynamic logic) formulae can significantly be reduced. SPDL is the stochastic extension of the multi-modal program logic PDL. SPDL provides means to specify path-based properties with or without timing restrictions. Paths can be characterised by so-called programs, essentially regular expressions, where the executability can be made dependent on the validity of test formulae. For model-checking SPDL path formulae it is necessary to build a product transition system (PTS) between the system model and the program automaton belonging to the path formula that is to be verified. In many cases, this PTS can be drastically reduced during the model checking procedure, as the program restricts the number of potentially satisfying paths. Therefore, we propose an approach that directly generates the reduced PTS from a given SPA specification and an SPDL path formula. The feasibility of this approach is shown through a selection of case studies, which show enormous state space reductions, at no increase in generation time.

1 Introduction

It is extremely important to develop techniques that allow the construction and analysis of distributed computer and communication systems. These systems must work correctly and meet high performance and dependability requirements. Using stochastic model checking it is possible to perform a combined analysis of both qualitative (correctness) and quantitative (performance and dependability) aspects of a system model. Models that incorporate both qualitative and quantitative aspects of system behaviour can be modelled by various high-level formalisms, such as stochastic process algebras [16, 15], stochastic Petri nets [1], stochastic activity networks [21] (SANs), etc.

In order to do model checking of stochastic systems, over the last years a number of stochastic extensions of the logic CTL [10] have been devised. The most notable extension is the logic CSL [4] (continuous stochastic logic). More recently, in [18, 3], action-based extensions of CSL were introduced. These logics allow for the specification of desired system behaviour by means of action sequences. This makes them very well suited for modelling formalisms in which

the actual system behaviour is specified as a sequence of actions or transitions, as is the case for SPAs, SPNs and SANs.

The applicability of stochastic model checking is limited by the complexity, i.e., the size of system models that are to be verified. At the heart of stochastic model checking lies the solution process of huge sparse sets of linear (differential) equations. This limits the size of systems that are practically analysable to some 10^8 states.

To overcome these limitations we can think of several approaches. One standard approach is the use of some notion of Markovian bisimulation. This approach has the following drawbacks. Computing the bisimulation quotient of a system is computationally expensive, and before reduction takes place the entire system has to be generated. Furthermore, depending on the system, the reduction in size may not be very large, and finally, due to reasons that are related to numerical analysis, the verification of the reduced system may be slower than that of the original system (cf. [17]).

We propose a different approach, which reduces the system size in many cases already during the state space generation, by exploiting the SPDL path formula that is to be verified.

Related Work For stochastic model checking we are not aware of any approach that generates the state space in a way which depends on the formula that is to be verified. For CSL model checking, in [4] an approach is described that makes states absorbing that do not functionally satisfy a given until-formula, but this state space reduction is performed only after the state space was generated. Following this proposal, in [18] model checking algorithms for SPDL path formulae were implemented. For CSL model checking this was done in [17]. For CTL model checking in [2] an approach is reported, where for interacting finite state machines equivalence relations are computed, depending on the CTL formula that is to be verified.

The paper is further organised as follows. In Section 2 we briefly introduce the syntax and semantics of SPDL; we will explain in an informal style the traditional approach to the model checking of SPDL path formulae. In Section 3 we then describe the stochastic process algebra $\mathcal{YAMPA}$, on which our property-driven state space generation approach relies. Section 5 is devoted to a denotational, symbolic property-driven semantics of $\mathcal{YAMPA}$. In Section 6 we will show the feasibility of our approach via some experimental results. Finally, Section 8 concludes the paper with a short summary and some pointers to future work.

2 SPDL - Syntax, Semantics and Model Checking

The logic SPDL is the stochastic extension of the logic PDL [12], a multi-modal program logic. PDL enriches the standard modal operator $\diamond$ ("possibly") with so-called programs, which are essentially regular expressions and tests (cf. Def. 1). In PDL, the formula $< \rho > \Phi$ means, that it is possible to execute program ρ and end in a state that satisfies Φ. SPDL adds the following extensions to PDL: The operator $< \rho >$ is replaced by the time-bounded path operator $[\rho]^I$, a

probability operator $\mathcal{P}_{\bowtie p}$ to reason about the transient system behaviour, and a steady state operator $\mathcal{S}_{\bowtie p}$ to reason about system behaviour, once stationarity has been reached. In what follows, we discuss the syntax, semantics, and a model checking procedure for SPDL.

2.1 Syntax of SPDL

Definition 1 (Syntax of SPDL). *Let p be a probability value in $[0,1]$, $q \in \mathsf{AP}$ an atomic proposition, where AP is the set of atomic propositions, and $\bowtie \in \{\leq, <, \geq, >\}$ a comparison operator. The state formulae Φ of SPDL are defined as:*

$$\Phi := q \big| \Phi \vee \Phi \big| \neg\Phi \big| \mathcal{P}_{\bowtie p}(\phi) \big| \mathcal{S}_{\bowtie p}(\Phi) \big| (\Phi)$$

Path formulae are defined as:

$$\phi := \Phi[\rho]^I \Phi,$$

where I is the closed interval $[t, t']$, Φ is assumed not to possess sub-formulae containing the steady state operator $\mathcal{S}_{\bowtie p}$.[1] *Programs ρ are described by the grammar given in Def. 2.*

Definition 2 (Programs). *Let Act be a set of actions, which are also called atomic programs, and TEST be a set of SPDL state formulae, again not containing the steady state operator $\mathcal{S}_{\bowtie p}$. A program ρ is defined by the following grammar:*

$$\rho := \epsilon \big| \Phi?; a \big| \rho; \rho \big| \rho \cup \rho \big| \rho^* \big| \Phi?; \rho \big| (\rho)$$

where $\epsilon \notin \mathsf{Act}$ is the empty program, $a \in \mathsf{Act}$ and $\Phi \in \mathsf{TEST}$.

Sequence ($;$), choice ($\cup$), and Kleene-star ($*$) have their usual meaning as known from the theory of regular expressions. The operator $\Phi?$ is the so-called test operator. Informally speaking, it tests whether Φ holds in the current state of the model. If this is the case, then execute program ρ, otherwise ρ is not executable. Following language theory, we can derive words from a program ρ (here also called program instances) according to the rules of regular expressions. The set of all these program instances is called a language.

Example 1. Throughout this paper, we use the example of a fault-tolerant packet collector, which has the following repeating behaviour. Arrivals can either be error-free (upper transition *arr*, rate λ) or erroneous (lower transition *error*, rate μ). If a data packet contains an error, this error can be correctable (*co*) non-correctable (*nco*). In case of a correctable error, the error is corrected (transition *co*) and more data packets can be received. If the error is non-correctable, the data packet has to be retransmitted (transition *rt*). In Fig. 1, the SLTS $\mathcal{M}$ for the packet collector is shown, where we assume that the number n of data

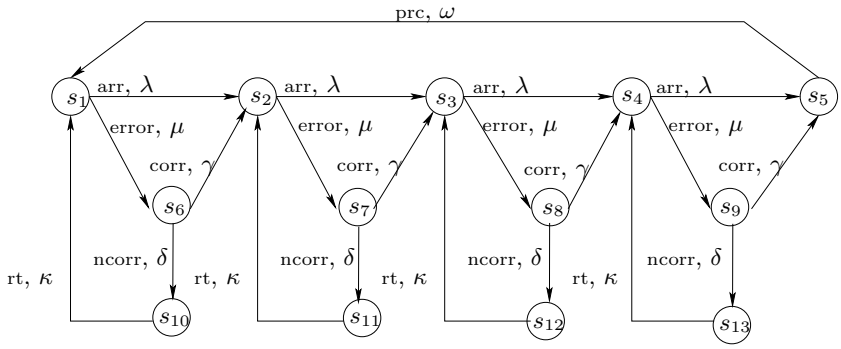

Fig. 1. Fault tolerant packet collector for $n = 4$ packets

packets that are to be processed is equal to four. The system has the following state labels:

$$L(s_5) = \{\mathsf{full}\}, \quad L(s_6) = ... = L(s_9) = \{\mathsf{error}\},$$
$$L(s_{10}) = ... = L(s_{13}) = \{\mathsf{waitrt}\}, \quad L(s_{14}) = ... = L(s_{17}) = \{\mathsf{waitcor}\}$$

The set of actions is given as follows:

$$\mathsf{Act} := \{arr, error, rt, corr, ncorr, prc\}$$

Using SPDL, we can easily express the following properties:

- $\Phi_1 := \mathcal{P}_{\bowtie p}((\neg\mathsf{full})[arr^*]^{[0,t]}(\mathsf{full}))$: Is the probability to receive N data packets without error within t time units greater or less than p?
- $\Phi_2 := \mathcal{P}_{\bowtie p}(\neg\mathsf{full}[arr; \mathsf{TEST1}?; error; rt; arr^* \cup arr^*]^{[0,t]}\mathsf{full})$: Is the probability to receive N data packets without error or with at most one non-correctable error within t time units greater or less than p, given that this non-correctable error appears in the first data packet? The test formula $\mathsf{TEST1}$ defines those states, in which it holds that 1 packet has arrived.
- $\Phi_3 := \mathcal{P}_{\bowtie p}(\mathsf{true}[arr^*; \mathsf{TEST2}?; arr; corr]^{[0,t]}\mathsf{full})$: Is the probability that the buffer is full after at most t time units and that the Nth packet contains a correctable error, given that all preceeding packets were error free, within the probability bounds given by $\bowtie\ p$? The test formula $\mathsf{TEST2}$ describes those states, in which it holds that $N - 1$ packets have arrived.

2.2 Semantics of SPDL

We will now show, both the model over which SPDL formulae are interpreted and the semantics of SPDL formulae.[2]

[1] Mixing formulae that express transient behaviour ($\mathcal{P}_{\bowtie p}$) with formulae expressing steady state behaviour ($\mathcal{S}_{\bowtie p}$) is considered less meaningful.

[2] The stochastic process algebra from Sections 3 and 5 and SPDL share the same semantic model.

The semantic model of SPDL is a so-called stochastic labelled transition system, defined as follows.

Definition 3 (Stochastic labelled transition system (SLTS)). *An SLTS $\mathcal{M}$ is a six-tuple $(s, S, \mathsf{Act}, L, R, \mathsf{AP})$, where*

- *s is the unique initial state,*
- *S is a finite set of states,*
- *Act is a finite set of action names,*
- *L is the state labelling function: $S \to 2^{AP}$,*
- *R is the state transition relation : $R \subseteq S \times (\mathsf{Act} \times \mathbb{R}_{>0}) \times S$,*
- *AP is the set of atomic propositions.*

Definition 4 (Semantics of SPDL).

- *The semantics of propositional logic formulae $\neg\Phi$ and $\Phi \vee \Psi$ is defined the usual way.*
- *$\mathcal{S}_{\bowtie p}(\Phi)$ asserts that the steady state probability of the Φ-states, i.e., the probability to reside in a Φ-state once the system has reached stationarity satisfies the probability bounds as given by $\bowtie p$.*
- *$\mathcal{P}_{\bowtie p}(\phi)$ asserts that the probability measure of all paths that satisfy ϕ lies within the bounds as imposed by $\bowtie p$.*
- *$\Phi[\rho]^I\Psi$ asserts that a path that satisfies this formula reaches a Ψ-state within at least t time units, but after at most t' time units. All preceeding states must satisfy Φ. Alternatively, a $\Phi \wedge \Psi$-state can be reached before the passage of t time units, but not left before at least t time units have passed. Additionally, the action sequence on the path to the Ψ-state must correspond to the action sequence of a word from the language induced by program ρ. All test formulae that are part of ρ must be satisfied by corresponding states of the path.*

2.3 Model Checking SPDL

The overall model checking algorithm of SPDL is similar to that of CTL, in the sense that it starts with the verification of atomic properties and then proceeds with the checking of ever more complex sub-formulae until the overall formula has been checked.

Model Checking SPDL

- Propositional formulae $\neg\Phi$ and $\Phi \vee \Psi$ are checked as in the CTL case.
- Steady state formulae $\mathcal{S}_{\bowtie p}(\Phi)$ can be checked as for CSL [4].
- Model checking formulae with a leading $\mathcal{P}_{\bowtie p}$ operator is more involved. We assume, we want to check whether in an SLTS $\mathcal{M}$ a state s satisfies $\mathcal{P}_{\bowtie p}(\phi)$, with $\phi = \Phi[\rho]^I\Psi$. The basic idea is to reduce the model checking problem of SPDL to one of CSL, which consists of deciding whether a continuous time Markov chain (CTMC) $\mathcal{M}^\times$ (to be constructed) and a state $s^\times$ in $\mathcal{M}^\times$ satisfies the CSL formula $\mathcal{P}_{\bowtie p}(\mathsf{F}^I\mathsf{succ})$. A path satisfies $\mathsf{F}^I\mathsf{succ}$, if within time interval I a state is reached that satisfies the atomic property succ. To reach this goal, we proceed as follows:

1. From the program ρ we derive a deterministic program automaton A_ρ, which is a variant of deterministic finite automata.[3]
2. Using the given SLTS $\mathcal{M}$ and the program automaton A_ρ we build a product Markov chain. $\mathcal{M}^\times$. The state space of $\mathcal{M}^\times$ is the product of $\mathcal{M}$ and A_ρ, i.e., its states are of the form (s_i, z_i), where s_i is a state of $\mathcal{M}$ and z_i a state of A_ρ. Additionally, $\mathcal{M}^\times$ possesses one new, absorbing state: the state $FAIL$.
   In $\mathcal{M}^\times$ a transition $(s_i, z_i) \xrightarrow{\lambda} (s_j, z_j)$ is kept, where λ is the rate of the transition from s_i to s_j, iff the following two constraints are satisfied:
   - (s_i, z_i) must satisfy Φ, this is the case iff s_i satisfies Φ.
   - Both s_i and z_i must be capable to perform the same action, and if the current action is associated with a test, then s_i must also satisfy this test.

   If one of these two constraints is violated, we have to introduce a transition $(s_i, z_i) \xrightarrow{\lambda} FAIL$ and delete transition $(s_i, z_i) \xrightarrow{\lambda} (s_j, z_j)$.
3. Finally, to compute the probability measure of the paths that satisfy ϕ we proceed as follows. All states (s_j, z_j) of $\mathcal{M}^\times$ for which s_j is a Ψ-state and z_j is an accepting state of A_ρ are replaced by the newly introduced absorbing success state $SUCC$, labelled with the special, newly introduced atomic state formula succ, thereby redirecting all incoming transitions from the old states to the new $SUCC$ state.
4. At this point, it is possible to check, whether $\mathcal{P}_{\bowtie p}(\Phi[\rho]^{[t,t']}\Psi)$ is functionally satisfiable: If in $\mathcal{M}^\times$ a path to a succ state exists, then $\mathcal{P}_{\bowtie p}(\Phi[\rho]^{[t,t']}\Psi)$ can be satisfied at least on the functional level.
5. On $\mathcal{M}^\times$ (which was transformed as described in step 3) we can compute the probability measure of all paths satisfying the CSL formula $\mathcal{P}_{\bowtie p}(\mathsf{F}^{[t,t']}\mathsf{succ})$, which is equal to the probability measure of the paths satisfying the original formula $\mathcal{P}_{\bowtie p}(\Phi[\rho]^{[t,t']}\Psi)$ in the original model $\mathcal{M}$.

3 Stochastic Process Algebras

In the past 15 years, a number of stochastic process algebras have been devised, such as PEPA [16] and TIPP [15]. Here, we use the stochastic process algebra $\mathcal{YAMPA}$ (yet another Markovian process algebra), that is used in the tool CASPA [20], which we use for our empirical studies. We will both give a formal account of $\mathcal{YAMPA}$, and we will also illustrate its most important operators by means of a small example.

3.1 Syntax, Semantics, and Equivalence for $\mathcal{YAMPA}$

Stochastic process algebras are an extension of functional process algebras in the same way as stochastic Petri nets form an extension of Petri nets to reason about the performance of a modelled system.

[3] For the derivation of A_ρ from program ρ we refer to [18] for a thorough discussion of this issue. As such this issue does not play a crucial role in understanding this paper.

The purpose of stochastic process algebras is threefold:

- stochastic process algebras shall provide a unified framework to reason both about functional and performance aspects of a system model
- they may help to integrate performance evaluation into early stages of system design
- they exploit the advantages of functional process algebras such as abstraction, constructivity, rigorous formal semantics etc.

In the last decade a number of stochastic process algebras have been developed such as MPA [8], EMPA [6], PEPA [16] and TIPP [14, 15].

Syntax of Stochastic Process Algebras We will now present the syntax of $\mathcal{YAMPA}$:

Definition 5 (Syntax of $\mathcal{YAMPA}$). *For a set of actions* Act*, let* $a \in \mathsf{Act}^*$ *and* $b \in \mathsf{Act}$*. Let* $L \subseteq \mathsf{Act}$ *be a set of visible actions, let* $\lambda \in \mathbb{R}^{>0}$ *be a rate, and let* $X \in Var$ *be a process variable.* $\mathcal{YAMPA}$ *is the language whose terms are given by the following grammar:*

$$P := \mathsf{stop} \mid \mathsf{exit} \mid (a, \lambda); P \mid P + P \mid P >> P \mid P[> P$$
$$P|[L]|P \mid P|||P \mid \mathsf{hide}\ b\ \mathsf{in}\ P \mid recX : Q \mid P[a/b]$$

Semantics of Stochastic Process Algebras The operators have the following informal meaning:

- $P := \mathsf{stop}$: describes deadlocking, i.e. inactive behaviour
- $P := \mathsf{exit}$: describes successful termination of a process
- $P := (a, \lambda); Q$: After an exponentially distributed delay, governed by rate λ, action a can be performed instantanously, afterwards the process behaves as Q.
- $P := Q + R$: Process P behaves either as Q or R.
- $P := Q >> R$: P describes the sequential composition of Q and R. At first, P behaves as Q, after successful termination of Q, which is denoted by action δ, P behaves like R.
- $P := Q[> R$: P describes interruption of process Q by R. The execution of Q can be interrupted by R (after any action performed by Q), if R takes over control, Q cannot be resumed. In case of stop or successful termination of Q, R is not executed.
- $P := Q|[L]|R$: Process P describes synchronous parallel composition of processes Q and R. Actions from L must be performed by both processes at the same execution step. Actions not in L can be performed by both processes independently. In order to preserve the the Markov property of the new process, the rates associated with the actions engaged in the synchronous transition must be adopted (also referred to as "synchronised"). In the case of $\mathcal{YAMPA}$ the synchronisation policy is multiplication[4].

[4] In other stochastic process algebras different policies are chosen

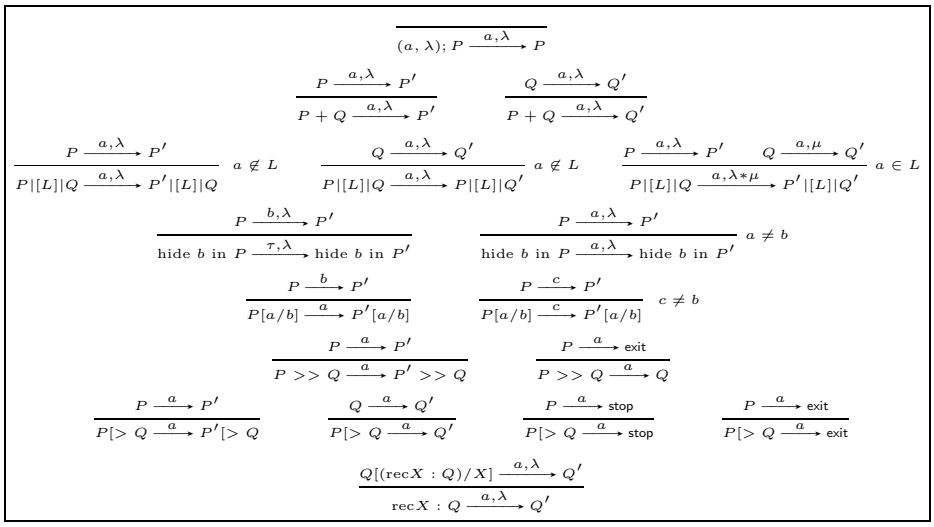

Fig. 2. Semantic rules for the language R-TIPP

- $P := Q|||R$: Process P describes asynchronous parallel composition of processes Q and R. The engaged processes can evolve independently of each other.
- $P :=$ hide a in Q: In P occurrences of action a are hidden from the environment, i.e. each occurrence of a in Q is replaced by the special silent action τ. The purpose of this operation is to prevent synchronisation over a on parallel composition of processes. One can interpret this as hiding of internal behaviour from the environment (abstraction).
- $P := recX : Q$: This operator describes cyclic, infinite behaviour of process P. In Q each occurrence of X is replaced by the definition of Q such that Q can be executed infinitely often. In practice, this is often described by having a process variable on both sides of a defining equation, for example:

$$P := a; b; P$$

  This means that process P can perform actions a followed by b infinitely often
- $P := Q[a/b]$: P describes relabelling, i.e. in process Q any occurrence of action b is replaced by a.

Formally, the semantics of $\mathcal{YAMPA}$ can be defined in SOS-style (cf. figure 2).

Example 2. In fig. 3 we list the $\mathcal{YAMPA}$ specification of the fault tolerant packet collector of example 1. In line (1) we can specify the maximum number of packets that must arrive, before processing starts. We see in this specification some "syntactic sugar" that eases the concise specification of complex systems, e.g., guarded choice in line (3). In lines (2) and (3) we find that process `Arr` is

```
(1)    int max = 15000;
(2)    System := Arr(0)|[error, corr,ncorr]| Errorhandler
(3)    Arr(i [max]) := [i=0] -> (arr, lambda);Arr(i+1) +
(4)                                    (error, mu);((corr, 1);Arr(i+1) + (ncorr,1);(rt,kappa);Arr(0))
(5)                    [i<max, i > 0] -> (arr, lambda);Arr(i+1) + (error, mu);((corr, 1);Arr(i+1) +
(6)                                    (ncorr,1);(rt,kappa);Arr(i-1))
(7)                    [i=max] -> (prc, omega);Arr(0)
(8)    Errorhandler := (error, 1);((corr, gamma);Errorhandler + (ncorr, delta);Errorhandler)
```

Fig. 3. Example $\mathcal{YAMPA}$ specification

parameterised with parameter `i`, that can take the maximum value `max`. This parameter records the number of packets that arrived. In line (2) we see that `Arr` is initialised with `i` = 0, i.e., zero packets arrived in the beginning.

The overall system consists of the processes `Arr` and `Errorhandler` that are composed in parallel and that have to synchronise over the actions `error, corr, ncorr`, i.e., these actions must be performed by both processes at the same time. For all other actions, the processes can evolve independently. `(arr, lambda);Arr(i+1)` (line (3)) is an example of prefix: After an exponentially distributed delay time, which is governed by rate `lambda`, action `arr` can be taken. In line (4) we find an example of choice: This process can either behave as `(arr, lambda);Arr(i+1)` or `(error, mu);((corr, 1);Arr(i+1) + (ncorr,1);(rt,kappa);Arr(0))`. In line (3) to (7) we see examples of guarded choice: Depending on the actual value of `i` different branches of the specification in lines (3) to (7) can be taken. In line (3), this branch of the specification can only be taken, if the value of parameter `i` is equal to zero. Process `Arr(i [max])` possesses cyclic (recursive) behaviour, as, after `arr` it can again behave as `Arr`.

4 Representing Transition Systems by Binary Decision Diagrams

In this section we will describe how stochastic labelled transition systems can be represented by means of MTBDDs. The notion of a path in an SLTS is defined as follows:

Definition 6 (Paths in SLTSs). *A finite path σ in an SLTS $\mathcal{T} = (S, \mathsf{Act}, \mathsf{AP}, \longrightarrow, s_0)$ is a sequence $s_0 \xrightarrow{a_1,t_1} s_1 \xrightarrow{a_2,t_2} s_2 \ldots s_{l-1} \xrightarrow{a_l,t_l} s_l$ with $l \in \mathbb{N}$, $s_i \in S$, $a_i \in \mathsf{Act}$. Such a path has length l and $\sigma[i] = s_i$ denotes its $(i{+}1)$-st state. Let $Path(s)$ denote the set of paths originating in s.*

Definition 7 (Transition encoding function). *A transition $x \xrightarrow{a,\lambda} y$ of an SLTS can be encoded using the minterm function:*

$$TR(x \xrightarrow{a,\lambda} y) \;:=\; MT(\vec{s}, Enc_S(x)) * MT(\vec{a}, Enc_{\mathsf{Act}}(a)) * MT(\vec{t}, Enc_S(y)) * \lambda$$

where $\vec{a}$ denotes the vector of Boolean variables encoding the action, and $\vec{s}$ and $\vec{t}$ denote the vectors of Boolean variables encoding the source and target state of the transition. In the sequel $TR(x \xrightarrow{a,\lambda} y)$ will be written as $TR(x, a, \lambda, y)$.

Definition 8 (Encoding function). *Let M be an arbitrary finite set. $Enc_M(m)$ denotes the injective encoding function that maps $m \in M$ to its binary encoding (a Boolean vector) of length n, i.e. $Enc_M : M \mapsto \mathbb{B}^n$, $n \geq \lceil \log_2 |M| \rceil$. If M is obvious from the context, the index of the encoding function can be omitted. We write $Enc_M(m) = \vec{m} = (m_{n-1}, \ldots, m_0)$.*

Definition 9 (Encoding sets). *Let the length n of an encoding be given. PC is the set of all possible binary encodings, i.e. $PC := \mathbb{B}^n$. The set of used encodings UC contains those elements of PC that were already used to encode elements of a given set M, i.e. $UC := \{\vec{c} \mid \vec{c} \in PC \wedge \exists m \in M : (Enc_M(m) = \vec{c})\}$. The set of free encodings FC contains those elements of PC that are not in UC, i.e. $FC := PC \setminus UC$.*

Definition 10 (Extension of a set of encodings by a leading binary digit). *Let C be a set of Boolean vectors of length n. $Ext_0(C)$ is obtained by adding a leading zero to the elements of C, i.e.:*

$$Ext_0(C) = \{\vec{c'} \mid \vec{c'} = 0 \circ \vec{c} \ \wedge \ \vec{c} \in C\}$$

Analogously we obtain $Ext_1(C)$ from C by adding a leading one. The function $Ext(C)$ adds an arbitrary leading digit to the vectors in C, i.e. $Ext(C) = Ext_0(C) \cup Ext_1(C)$.

Definition 11 (Choice of encoding). *An element $\vec{c}$ of a given set of encodings C is chosen with respect to a total ordering relation $\bowtie$ by the function $Ch(C, \bowtie) := \vec{c} \in C$ such that $\forall \vec{c'} \in C : (\vec{c} \bowtie \vec{c'})$.*

In the sequel we assume that the BDD variables have the following ordering, denoted by $\prec$:

At the first $n_a \geq \lceil \log_2 |\mathsf{Act}| \rceil$ levels from the root are the variables a_i encoding the action. On the remaining levels we have $2 * n_s \geq 2 * \lceil \log_2 |S| \rceil$ variables encoding the source and target state of a transition. The source state variables (s_i) and the target states variables (t_i) are ordered in an interleaved fashion, which yields the following overall variable ordering[5]:

$$a_0 \prec \ldots \prec a_{n_a-1} \prec s_0 \prec t_0 \prec \ldots \prec s_{n_s-1} \prec t_{n_s-1}$$

Example 3. Let the SLTS from figure 4 be given. This system has the actions arr, $serve$, $fail$, $repair$, therefore we need two binary variables, a_1 and a_0, to encode them binarily:

$$\begin{aligned} Enc(arr) &= 00 \\ Enc(serve) &= 01 \\ Enc(fail) &= 10 \\ Enc(repair) &= 11 \end{aligned}$$

[5] This interleaved ordering is the commonly accepted heuristics for obtaining small MTBDD sizes, see for instance [11, 13, 22].

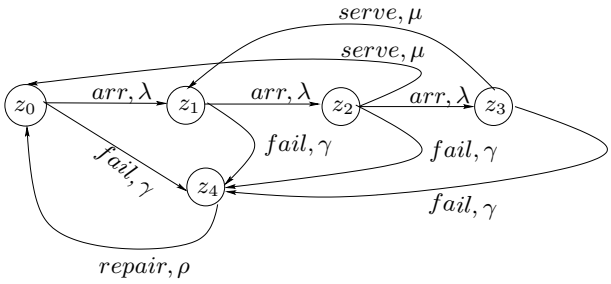

Fig. 4. Example SLTS

The five states are encoded as follows:

$$\begin{aligned} Enc(z_0) &= 000 \\ Enc(z_1) &= 001 \\ Enc(z_2) &= 011 \\ Enc(z_4) &= 010 \\ Enc(z_3) &= 100 \end{aligned}$$

For example the transition $z_1 \xrightarrow{fail} z_4$ is encoded binarily using function TR as follows:

$$\begin{aligned} TR(z_1 \xrightarrow{fail,\gamma} z_4) = MT(\mathbf{s}, z_1) * MT(\mathbf{a}, fail) * MT(\mathbf{t}, z_4) * \gamma = \\ MT(\mathbf{s}, 000) * MT(\mathbf{a}, 10) * MT(\mathbf{t}, 010) = \\ \neg s_0 * \neg s_1 * \neg s_2 * a_0 * \neg a_1 * \neg t_0 * t_1 * \neg t_2 \end{aligned}$$

In figure 5 we find the final MTBDD representation of the given SLTS (cf. figure ⊠

5 A Property-Driven Symbolic Semantics for $\mathcal{YAMPA}$

In this section we introduce the new property-driven semantics for $\mathcal{YAMPA}$. In Sec. 5.1 we will give the general idea of this semantics. (MTBDDs) as data structure to represent SLTSs. In Sec 5.2 formal definition of the new property-aware semantics of $\mathcal{YAMPA}$ is given. In Sec. 5.3 the semantic rules are illustrated by means of a small example

5.1 General Idea

In Section 2.3 we have presented a straight-forward model checking procedure for SPDL path formulae. The size of the product CTMC, before it is reduced is the product of the sizes of the original model $\mathcal{M}$ and the program automaton A_ρ. During the model checking procedure, many states are merged into the states

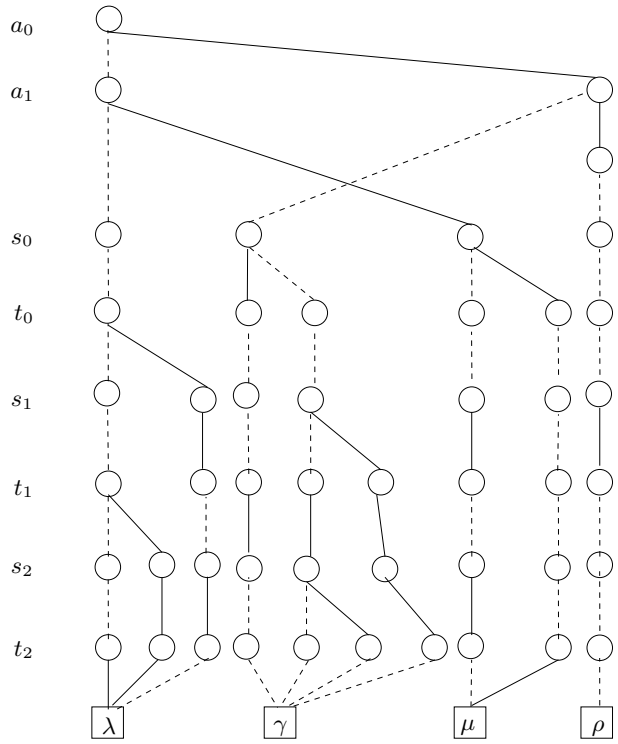

Fig. 5. Final MTBDD representation of SLTS

$FAIL$ resp. $SUCC$. This means, we needlessly generate a state space that is much larger than actually required, which is both a waste of memory space and time.

To overcome this weakness in the usual model checking procedure we propose an approach that generates only those states that are actually needed to verify the property at hand. In order to reach this goal, we introduce a property-driven semantics for the stochastic process algebra $\mathcal{YAMPA}$, that uses the path formula that is to be verified to direct the state space generation process. This new semantics cuts off state space generation as soon as it becomes clear a path is either not satisfying, i.e., it leads to a $FAIL$ state, or satisfying, i.e., leads to a $SUCC$ state. This significantly reduces the number of states and transitions that are generated.

We will use the symbolic semantics of [19] as a basis for our new SPA semantics. Like in [19], the property-driven semantics maps the SPA specification directly to the MTBDD representation of its underlying SLTS. The semantics proceeds in a compositional manner, according to the syntactic structure of the process term at hand. Additionally to [19], the new semantics takes, as already said, during generation of the SLTS the SPDL property that is to be verified into account. We chose MTBDDs as data structures for the SLTS representation

as it was shown convincingly [22] that MTBDDs allow a compact representation of even huge state spaces.

Definition 12 (Symbolic representation of process algebra terms). *The symbolic representation $[\![P]\!]$ of a process algebra term P consists of the following parts:*

- *The MTBDD $B(P)$, encoding the transition relation,*
- *a list of encodings of process variables X, that appear in P, denoted $Enc_S(X)$* [6]*,*
- *the encoding of the initial state of P, denoted $Enc_S(s_P^{DS})$,*
- *the transition relation δ_{A_ρ} for A_ρ,*
- *the current state of A_ρ.*

The list of action encodings $Enc_{\mathsf{Act}}(a)$ is globally valid for all processes and therefore not included in $[\![P]\!]$. In the following we describe how to obtain $[\![P]\!]$ from the symbolic representations of its constituents.

Parallel Transitions Two transitions $s_1 \xrightarrow{a_1,\lambda_1} t_1$ and $s_2 \xrightarrow{a_2,\lambda_2} t_2$ are called parallel if $s_1 = s_2$ and $a_1 = a_2$ and $t_1 = t_2$ (note that, in principle, both $\lambda_1 \neq \lambda_2$ and $\lambda_1 = \lambda_2$ is possible, although the latter case is ruled out if we only consider ordinary transition systems, as opposed to multi-transition systems). Parallel transitions can be created by applying the choice or hiding operators, or by applying the recursion operator in combination with choice. As we will see, our MTBDD semantics does not represent parallel transitions separately, but cumulates their rates, which is correct by lemma 1.

Lemma 1 (Cumulation of parallel transitions). *Let $\mathcal{T}$ be an SLTS and let transition system $\mathcal{T}'$ be constructed from $\mathcal{T}$ by cumulating parallel transitions, i.e. by replacing each set of parallel transitions $\{s \xrightarrow{a,\lambda_i} t \mid i = 1,\ldots,n\}$ by a single transition $s \xrightarrow{a,\lambda} t$, where $\lambda = \sum_{i=1}^{n} \lambda_i$. Then $\mathcal{T} \sim_M \mathcal{T}'$.*

The proof is straight-forward by comparing the cumulative rates, the details are omitted.

5.2 Property-Driven Symbolic Semantics - Formal Definition

In the sequel, we will give for every operator of the process algebra $\mathcal{YAMPA}$ the formal rule of the property-aware symbolic semantics.

[6] Process variables correspond to states, therefore we use Enc_S for both states and process variables.

(1) **if** not first appearance of X:
(2) **skip**
(3) **endif**
(4) **if** $FC := \emptyset$
(5) $PC := Ext(PC)$; $UC := Ext_0(UC)$; $PC := FC \backslash UC$
(6) **endif**
(7) $Enc_S(X) := Ch(FC, <)$;
(8) $B(P) := 0$

Fig. 6. Algorithm for process variable X and stop

(1) **if** not first appearance of stop:
(2) **skip**
(3) **endif**
(4) **if** $FC := \emptyset$
(5) $PC := Ext(PC)$; $UC := Ext_0(UC)$; $PC := FC \backslash UC$
(6) **endif**
(7) $Enc_S(\mathsf{stop}) := Ch(FC, <)$;
(8) $B(P) := 0$

Fig. 7. Algorithm for stop

Process Variables *X* A process variable specifies a reference state within a surrounding $recX$-operator. Therefore, process variables are encoded in a similar fashion as states, i.e. their encodings can be taken from $P.C.$ (the set of possible encodings). Within each sequential component, process variables having the same name get the same encoding. Upon first appearance of X, the MTBDD associated with X is the 0-MTBDD, that is, the MTBDD consisting only of the 0 terminal vertex.
Formal Description: See fig. 6

Stop Process: *P* := stop The stop-process is a special case of a process variable, a process constant. The stop-process has no emanating behaviour and remains inactive forever. As in the case of process variables, the stop-process is associated with the 0-MTBDD.
Formal Description: See fig. 7

Exit Process: *P* := exit Here, we will apply an adopted interpretation of the standard exit-process of LOTOS [7], that suffices in our context. Te exit-process is treated like the stop-process and has therefore no emanating behaviour, which means, that unlike in the standard LOTOS-interpretation, we do not need the special δ-action that normally expresses termination.
Formal Description: See fig. 8

(1) **if** not first appearance of exit:
(2) **skip**
(3) **endif**
(4) **if** $FC := \emptyset$
(5) $PC := Ext(PC)$; $UC := Ext_0(UC)$; $PC := FC \backslash UC$
(6) **endif**
(7) $Enc_S(\mathsf{exit}) := Ch(FC, <)$;
(8) $B(P) := 0$

Fig. 8. Algorithm for exit

Prefix $P := (a, \lambda); Q$ For a given formula $\Psi := \mathcal{P}_{\bowtie p}(\Phi_1[\rho]^I \Phi_2)$, we want to generate the symbolic representation of P, $[\![P]\!]$. To construct $B(P)$ we have to distinguish the following cases:

1. If the current state s_P^{DS} satisfies Φ_1 and in A_ρ's current state z an a-labelled transition to a state z' is possible, s_P^{DS} satisfies the test formula Ξ, possibly attached to A_ρ's a-transition, then, we can introduce a transition from s_P^{DS} to the encoding of Q's initial state.
2. If, additionally to case 1, the target state of A_ρ is an accepting state and s_Q^{DS} satisfies Φ_2, then a transition from the encoding of s_P^{DS} to the encoding of state $SUCC$[7] is introduced.
3. If the state s_P^{DS} satisfies Φ_1, but no transition labelling in A_ρ's current state matches a, then we have to introduce a transition from the encoding of P to the encoding of the error state $FAIL$.
4. If state s_P^{DS} does not satisfy Φ_1, then we have to introduce a transition from the encoding of P to the encoding of the error state $FAIL$.
5. If state s_P^{DS} does not satisfy the test formula, attached to A_ρ's a transition, then we have to introduce a transition from the encoding of P to the encoding of the error state $FAIL$.

Formal Description: See fig. 9

Choice $P := Q + R$ Here, we can assume, that $[\![Q]\!]$ and $[\![R]\!]$ are already available. To derive $[\![P]\!]$ from $[\![Q]\!]$ and $[\![R]\!]$ we can proceed as follows:

- A new initial state s_P^{DS} has to be introduced.
- All transitions emanating from the old initial states of Q resp. R have to be copied to s_P^{DS}.

Formal Description: See fig. 10

[7] $SUCC$ can be handled like stop.

Case 1:
(1) **if** $((s_P^{DS} \models \Phi_1) \wedge (z \xrightarrow{a}_{A_\rho} z') \wedge (s^{DS} \not\models \Phi_2) \wedge (z_\pi \notin E(A_\rho)))$
(2) **if** $FC = \emptyset$
(3) $PC := Ext(PC)$; $UC := Ext_0(UC)$; $FC := PC \backslash UC$
(4) **endif**
(5) $Enc_S(s_P^{DS}) := Ch(FC, <)$
(6) **if** $FC = \emptyset$
(7) $PC := Ext(PC)$; $UC := Ext_0(UC)$; $FC := PC \backslash UC$
(8) **endif**
(9) $Enc_S(s_Q^{DS}) := Ch(FC, <)$
(10) $B(P) := TR(s_P^{DS}, a, \lambda, s_Q^{DS})$
(11) **endif**
Case 2:
(1) **if** $((s_P^{DS} \models \Phi_1) \wedge (z \xrightarrow{a}_{A_\rho} z') \wedge (s_P^{DS} \models \Xi)(z' \in E(A_\rho)) \wedge (s_Q^{DS} \models \Phi_2))$
(2) **if** $FC = \emptyset$
(3) $PC := Ext(PC)$; $UC := Ext_0(UC)$; $FC := PC \backslash UC$
(4) **endif**
(5) $Enc_S(s_P^{DS}) := Ch(FC, <)$
(6) **if** $FC = \emptyset$
(7) $PC := Ext(PC)$; $UC := Ext_0(UC)$; $FC := PC \backslash UC$
(8) **endif**
(9) $Enc_S(s_{SUCC}^{DS}) := Ch(FC, <)$
(10) $B(P) := TR(s_P^{DS}, a, \lambda, SUCC)$
(11) **endif**
Case 3:
(1) **if** $((s_P^{DS} \models \Phi_1) \wedge (s_P^{DS} \models \Xi) \wedge (z \not\xrightarrow{a}_{A_\rho} z'))$
(2) **if** $FC = \emptyset$
(3) $PC := Ext(PC)$; $UC := Ext_0(UC)$; $FC := PC \backslash UC$
(4) **endif**
(5) $Enc_S(s_P^{DS}) := Ch(FC, <)$
(6) **if** $FC = \emptyset$
(7) $PC := Ext(PC)$; $UC := Ext_0(UC)$; $FC := PC \backslash UC$
(8) **endif**
(9) $Enc_S(s_{FAIL}^{DS}) := Ch(FC, <)$
(10) $B(P) := TR(s_P^{DS}, a, \lambda, FAIL)$
(11) **endif**
Case 4:
(1) **if** $(s^{DS} \not\models \Phi_1)$
(2) **if** $FC = \emptyset$
(3) $PC := Ext(PC)$; $UC := Ext_0(UC)$; $FC := PC \backslash UC$
(4) **endif**
(5) $Enc_S(s_P^{DS}) := Ch(FC, <)$
(6) **if** $FC = \emptyset$
(7) $PC := Ext(PC)$; $UC := Ext_0(UC)$; $FC := PC \backslash UC$
(8) **endif**
(9) $Enc_S(s_{FAIL}^{DS}) := Ch(FC, <)$
(10) $B(P) := TR(s_P^{DS}, a, \lambda, FAIL)$
(11) **endif**
Case 5:
(1) **if** $(s^{DS} \not\models \Xi)$
(2) **if** $FC = \emptyset$
(3) $PC := Ext(PC)$; $UC := Ext_0(UC)$; $FC := PC \backslash UC$
(4) **endif**
(5) $Enc_S(s_P^{DS}) := Ch(FC, <)$
(6) **if** $FC = \emptyset$
(7) $PC := Ext(PC)$; $UC := Ext_0(UC)$; $FC := PC \backslash UC$
(8) **endif**
(9) $Enc_S(s_{FAIL}^{DS}) := Ch(FC, <)$
(10) $B(P) := TR(s_P^{DS}, a, \lambda, FAIL)$
(11) **endif**

Fig. 9. Algorithm for prefix $P := (a, \lambda); Q$

(1) **if** $FC = \emptyset$
(2) $\quad PC := Ext(PC)$; $UC := Ext_0(UC)$; $FC := PC \backslash UC$
(3) **endif**
(4) $Enc_S(s_P^{DS}) := Ch(FC, <)$
(5) $B(Q') := B(Q)\big|_{\vec{s}=Enc_S(s_Q^{DS})} * MT(\vec{s}, Enc_S(s_P^{DS})$
(6) $B(R') := B(R)\big|_{\vec{s}=Enc_S(s_R^{DS})} * MT(\vec{s}, Enc_S(s_P^{DS})$
(7) $B(P) := B(Q) + B(R) + B(Q') + B(R')$

Fig. 10. Algorithm for choice $P := Q + R$

(1) $B(Q') := B(Q) * (1 - MT(\vec{t}, Enc_S(\mathsf{exit}))) + B(Q)\big|_{\vec{t}=Enc_S(\mathsf{exit})}$
(2) **if** $B(Q') \neq \emptyset$
(3) $\quad Enc_S(s_P^{DS}) := Enc_S(s_Q^{DS})$
(4) $\quad$ **if** $FC = \emptyset$
(5) $\quad\quad PC := Ext(PC)$; $UC := Ext_0(UC)$; $FC := PC \backslash UC$
(6) $\quad$ **endif**
(7) $\quad Enc_S(s_R^{DS}) := Ch(FC, <)$
(8) $\quad B(P) := B(Q') * MT(\vec{t}, Enc_S(s_R^{DS})$
(9) **endif**
(10) **else**
(11) $B(P) := B(Q)$

Fig. 11. Algorithm for enabling $P := Q >> R$

Enabling $P := Q >> R$ We can assume that $[\![Q]\!]$ is already available. To generate $[\![P]\!]$ from $[\![Q]\!]$ (and $[\![R]\!]$), we can proceed as follows:

- If in $B(Q)/[\![Q]\!]$ transitions to the encoding of the exit-state exist, they can be redirected to the newly introduced encoding of the intial state of R.
- If such transitions are possible, we can continue with the generation of the symbolic representation of R, $[\![R]\!]$. If no such transitions do exist, we are done and $[\![P]\!] = [\![R]\!]$.

Formal Description: See fig. 11

Disabling $P := Q[> R$ Generally spoken, disabling (or interruption) can be interpreted as repeated choice: In every state of Q it is possible that either Q's actual transitions are taken or that any of R's initial transitions is taken, having the appropriate successor in R as their target state. If any of R's transition is taken, Q cannot be resumed. Again, we can assume that the symbolic representation of Q, $[\![Q]\!]$ is already available. To generate $[\![P]\!]$ from $[\![Q]\!]$ and $[\![R]\!]$, we can proceed as follows:

- In any state of Q, transitions of R's initial state have to be added.
- To determine the 'appropriate' target states of R's transitions we have to distinguish the same cases as for the prefix operator.

(1) **forall** s_Q
(2) **if** $(Enc_S(s_Q) \neq Enc_S(\mathsf{exit}) \wedge Enc_S(s_Q) \neq Enc_S(\mathsf{stop}) \wedge Enc_S(s_Q) \neq Enc_S(SUCC) \wedge Enc_S(s_Q) \neq Enc_S(FAIL))$
(3) **forall** $(a \in Act(R, s_R^{DS})$ /*$Act(R, s_R^{DS})$ all in initial state of R active actions */
(4) **if** $((s_R^{DS} \models \Phi_1) \wedge (s_R^{DS} \models \Xi) \wedge (z \xrightarrow{\Xi?;a} z'))$
(5) $B(Q') := TR(s_Q, a, \lambda succ(a, s_R^{DS}))$
(6) **endif**
(7) **else**
(8) $B(Q') := TR(s_Q, a, \lambda, FAIL)$
(9) $B(Q) := B(Q')$
(10) **endforall**
(11) **endif**
(12) **endforall**

Fig. 12. Algorithm for disabling $P := Q[> R$

Here, we will give only case 1, the rest is similar to the prefix case.
Formal Description: See fig. 12

Parallel Composition $P := Q|[L]|R$ To derive $[\![P]\!]$ from Q and R and $\Phi_1[\rho]^I\Phi_2$, we must not assume that $[\![Q]\!]$ and $[\![R]\!]$ are already available.

Instead, we have to derive $[\![P]\!]$ from Q and R step by step, by respecting the same conditions as for the prefix operator, i.e., depending on the current state of s_P^{DS} of P, and z of A_ρ, we add transitions either to a "regular" successor of s_P^{DS} or to $FAIL$, resp. $SUCC$.

In Fig. 13 the algorithm for the derivation of $[\![P]\!]$ from Q and R for a single transition is given. We list only a few of the possible cases. This procedure has to be repeated, until all potential transitions that are possible are generated. This can be done using standard depth- or breadth-first search applied to P's parse tree.
Formal Description: See fig. 13 The remaining cases are quite similar, therefore we will omit them here. If $a \in L$, both target states of Q and R have to be taken into consideration. I.e. the conditions in line (2) must apply to both s_Q^{DS} and s_R^{DS}.

Hiding $P :=$ hide a in$Q$ For the derivation of $[\![P]\!]$ we must take the syntactic structure of Q into account, i.e. no general formal algorithm for this operator can be given, as the derivation of $[\![P]\!]$ depends on Q. The only thing that can be said, is that within the scope of the hiding operator all conditions w.r.t. the action name must be applied to τ instead of a.

Relabelling $P := Q[a/b]$ For the derivation of $[\![P]\!]$ we must take the syntactic structure of Q into account, i.e. no general formal algorithm for this operator can be given, as the derivation of $[\![P]\!]$ depends on Q. The only thing that can

(1) **while** P's parse tree NOT fully explored:
(2) **if** $((a \notin L) \wedge (s_Q^{DS} \models \Phi_1) \wedge (s_Q^{DS} \models \Xi) \wedge (z \xrightarrow{\Xi?;a}_{A_\rho} z'))$
(3) **if** $FC = \emptyset$
(5) $PC := Ext(PC)$; $UC := Ext_0(UC)$; $FC := PC \backslash UC$
(6) **endif**
(7) $Enc_S(s_Q^{DS}) := Ch(FC, <)$
(8) **if** $FC = \emptyset$
(9) $PC := Ext(PC)$; $UC := Ext_0(UC)$; $FC := PC \backslash UC$
(10) **endif**
(11) $Enc_S(s_R^{DS}) := Ch(FC, <)$
(12) $Enc_S(s_P^{DS}) := Enc_S(s_Q^{DS}) \circ Enc_S(s_R^{DS})$
(13) **if** $FC = \emptyset$
(14) $PC := Ext(PC)$; $UC := Ext_0(UC)$; $FC := PC \backslash UC$
(15) **endif**
(16) $Enc_S(s_{Q'}^{DS}) := Ch(FC, <)$
(17) $Enc_S(s_{P'}^{DS}) := Enc_S(s_{Q'}^{DS}) \circ Enc_S(s_R^{DS})$
(18) $B(Q') := TR(s_P^{DS} a, \lambda, s_{P'}^{DS}$
(19) $B(P) := B(P) + B(Q')$
(20) **endif**
(21) **endwhile**

Fig. 13. Algorithm for parallel composition $P := Q|[L]|R$

be said, is that within the scope of the hiding operator all conditions w.r.t. the action name must be applied to a instead of b.

5.3 Example of Property-Driven Semantics

Example 4. We want to generate the SLTS for the specification from Example 2, with `max = 2`. and SPDL formula $\Phi_1 := \mathcal{P}_{\bowtie p}((\neg \mathsf{full})[arr^*]^{[0,t]}(\mathsf{full}))$ from Example 1. We assume, that the actions and their encodings are globally known, i.e., we know the number of Boolean variables required for their encoding, which is three. As we derive the MTBDD representation of the SLTS directly from the given specification we do not know in advance the size of the state space and therefore the number of Boolean variables to encode the states and the transition relation. Therefore, we take in the beginning as small a number as possible, and extend the number of variables, if required. The initial state of the specification `Arr(0)|[error, corr,ncorr]| Errorhandler` can be encoded by one Boolean variable $Enc_S(s_1) = \neg z_1 = 0$. Given Φ_1, we check if $\neg$full is satisfied, which is the case, then we check whether a transition labelled with arr is possible, which is the case, i.e., we add $Enc_S(s_2) = z_1 = 1$. As for s_2 the condition full is not satisfied, $s_2 \neq SUCC$. The MTBDD encodes at this point the transition relation R consisting of $TR(s_0, arr, \lambda, s_1)$. In s_1 a second transition, labelled by $error$ is possible, we see from Φ_1 that err does not belong to the actions that yield a satisfying path, i.e., we have to introduce a transition to the failure state $FAIL$,

which has no encoding up to now. To do so, we have to extend the number of Boolean variables that encode states, i.e., the states s_1 and s_2 are re-encoded:

$$Enc_S(s_1) = \neg z_2 \wedge z_1 = 00 \;\; Enc_S(s_2) = \neg z_2 \wedge z_1 = 01$$
$$Enc_S(FAIL) = z_2 \wedge \neg z_1 = 10$$

Now, we can introduce a new transition encoding: $TR(s_1, error, \mu, FAIL)$. The overall transition relation R is now the disjunction of $TR(s_0, arr, \lambda, s_1)$ and $TR(s_1, error, \mu, FAIL)$

The state s_2 corresponds to `Arr(1)|[error, corr,ncorr]| Errorhandler`, i.e., $\neg$full is satisfied, and again *arr* and *error* transitions are possible, due to the restrictions imposed by the path formula, *error* leads to the $FAIL$ state, i.e., we introduce a new transition: $TR(s_2, error, \mu, FAIL)$. For *arr* we add a new transition from s_2 to s_3, as s_3 satisfies full, $s_3 = SUCC$, and $TR(s_2, arr, \lambda, SUCC)$, where $Enc_S(s_3) = 11$. In Fig. 14 we find the MTBDD encoding the transition relation of this SLTS.

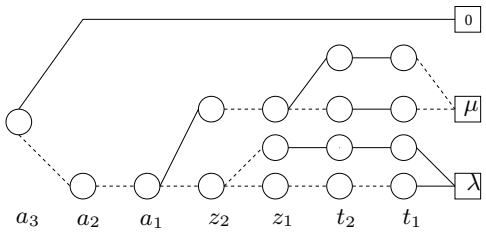

Fig. 14. MTBDD representation of the fault-tolerant packet collector's SLTS for `max` = 2 and Φ_1

6 Empirical Results

For our case studies we have employed the symbolic stochastic model checker CASPA. All results have been computed on a standard PC with Pentium IV 3.2 GHz processor, 1 GB RAM, running the operating system SuSe Linux 10.0.

6.1 Fault-Tolerant Packet Collector

Let us consider the system from Example 1. We will check the SPDL path formulae presented there. In Table 1 we find the model sizes for these formulae. In columns three to five, we list the maximum size of the product CTMC that is generated for model checking SPDL without property-driven state space generation, which is the product of the size of the automaton and the system model. In columns six to eight we list the state space sizes as they are generated when using the property-driven approach proposed in this paper, and on which model checking is actually carried out. We see, that we can avoid the generation of

many states, thereby reducing the memory requirements for SPDL model checking. We see in Table 2 that for both formulae the property-driven state space generation also requires less time than the traditional approach.

`max`	State space size	Not Property-driven			Property-driven		
		Φ_1	Φ_2	Φ_3	Φ_1	Φ_2	Φ_3
5,000	15,001	15,001	60,004	45,003	5,002	20,003	10,003
15,000	45,001	45,001	180,004	135,003	15,002	60,003	30,003
30,000	90,001	90,001	360,004	270,003	30,002	120,003	60,003
50,000	150,001	150,001	600,004	450,003	50,002	200,003	100,003

Table 1. State space sizes for Φ_1 to Φ_3 (Packet collector)

6.2 Kanban System

The Kanban manufacturing system was first described as a stochastic Petri net in [9]. We consider a Kanban system with four cells, a single type of Kanban cards and the possibility that some workpieces may need to be reworked. We will check the following properties:

- Φ_1: Is the requirement, that within t time units exactly three reworks are required in station 1 satisfied with a probability that is at most p?
- Φ_2 :: Is the probability that a single job needs at most t time units to go through all 4 stations greater than p percent?
- Φ_3: Is the probability to reach station 4, within t time units, given in station 1 are no reworks required and in stations 2 and 3 in total exactly 2 reworks are necessary within $\bowtie p$?

From Table 3 we observe that for the formulae Φ_1 to Φ_3 the state space of the product CTMC is dramatically smaller than that of the original system, which stems from the fact that for all three formulae only very specific paths in the system are of interest. We can observe that for Φ_2 the size of the product CTMC is independent of the number of Kanban cards, which is not surprising, as we consider a specific card that goes through the system. In the second column we find the size of the original state space, in columns three to five we show the maximum size of the state space for the traditional approach, and in columns six through eight we list the final state space on which model checking actual is performed. We see in Table 4 for all three formulae that property-driven state space generation requires less time than the traditional approach. This is not surprising, as billions of states and even more important, billions of transitions of the original model do not to be explored in the property-driven approach.

6.3 Fault-Tolerant Multiprocessor System

This example is based on [21]. The original model consists of N computers each of which has the following components: Memory modules, CPUs, I/O ports, and

max	Not Property-driven			Property-driven		
	Φ_1	Φ_2	Φ_3	Φ_1	Φ_2	Φ_3
5,000	2.9 sec.	3.3 sec.	3.1 sec.	2.0 sec.	2.8 sec.	2.9 sec.
15,000	10.00 sec.	10.8 sec.	11.2 sec.	6.9 sec.	9.0 sec.	9.0 sec.
30,000	21.4 sec.	22.7 sec.	22.5 sec.	17.8 sec.	18.9 sec.	19.6 sec.
50,000	37.9 sec.	45.3 sec.	44.4 sec.	33.6 sec.	40.4 sec.	32.8 sec.

Table 2. State space generation times for Φ_1 to Φ_3 (Packet collector)

n	State space size	Not Property-driven			Property-driven		
		Φ_1	Φ_2	Φ_3	Φ_1	Φ_2	Φ_3
5	2,546,432	22,917,888	33,103,616	43,289,344	83	13	159
8	133,865,325	1,204,787,925	1,740,249,225	2275710525	189	13	240
10	1,005,927,208	9,053,344,872	13,077,053,704	17,100,762,536	276	13	294
12	5,519,907,575	49,679,168,175	71,758,798,475	93,838,428,775	364	13	348
15	46,998,779,904	-	-	-	496	13	411

Table 3. State space sizes for Φ_1 and Φ_2 (Kanban)

n	Not Property-driven			Property-driven		
	Φ_1	Φ_2	Φ_3	Φ_1	Φ_2	Φ_3
5	0.8 sec.	0.7 sec.	0.7 sec.	0.1 sec.	0.1 sec.	0.1 sec.
8	4.7 sec.	4.2 sec.	4.5 sec.	0.2 sec.	0.2 sec.	0.2 sec.
10	11.4 sec.	10.8 sec.	11.0 sec.	0.5 sec.	0.5 sec.	0.5 sec.
12	21.7 sec.	21.5 sec.	22.1 sec.	0.8 sec.	0.7 sec.	0.7 sec.
15	-	-	-	1.6 sec.	1.5 sec.	1.5 sec.

Table 4. State space generation times for Φ_1 and Φ_2 (Kanban)

error handlers. Each of these computer components consists of several subcomponents, that can fail, leading to the failure of one computer. The overall system is operational if at least one computer is operational.

We have generated the CTMC for three different configurations: C1 is the configuration consisting of two computers with three memory modules each; C1 has about 750,000 reachable states. C2 consists of 3 computers, with one memory module each. C3 comprises 3 computers and 3 memory modules each.

We will check the following formula Φ_1: Does the probability that computer failures and subsequently a system failure is only due to memory failures lie within the bounds as given by $\bowtie p$, given that the maximum time to reach a system failure state is at most t?

In Table 5 we show the model sizes for the above formulae. In column three, we list the maximum size of the product CTMC that is generated for model checking SPDL without property-driven state space generation, which is the product of the size of the automaton and the system model. In column 4 we give the model size, when applying the property-driven state space generator. We do not list the model generation times here, which are below 0.1 sec. for all configurations, in both the property-driven and the non-property-driven case.

Conf	State space size	Not Property-driven	Property-driven
		Φ_1	Φ_1
C1	753,664	2,260,992	53,306
C2	123,760	371,280	1,475
C3	381,681,664	1,145,044,992	6,554,329

Table 5. State space generation times for Φ_1 (fault-tolerant multi-processor)

7 Correctness of the Property-Aware Semantics

The following theorem states that the property aware semantics is correct:

Theorem 1 (Correctness of Property-Aware Semantics). *The property aware semantics is correct. I.e. the probability of satisfying $\Phi[\rho]^{[0,t]}\Psi$ in $\mathcal{M}$ is equal to the probability of reaching $\chi_{\mathcal{M}^\times}$ within time $t' \in [0,t]$ in $\mathcal{M}^\times$:*

$$Pr\{\sigma \in Path_s^{\mathcal{M}} \mid \mathcal{M}, \sigma \models \Phi[\rho]^{[0,t]}\Psi\} =$$
$$Pr\{\sigma^\times \in Path_{(s,z_0)}^{\mathcal{M}^\times} \mid \exists t \in [0,t'] : \mathcal{M}^\times, \sigma^\times @ t \models \chi_{\mathcal{M}^\times}\}$$

] Before we can prove theorem 1 we need the following definitions:

Definition 13 (Indicator function). *The function $Ind(\mathcal{M}, s, \phi)$ indicates, whether an arbitrary SPDL state formula ϕ is satisfied in a given state s of a fixed model $\mathcal{M}$:*

$$Ind(\mathcal{M}, s, \phi) = \begin{cases} 1 & \textit{iff } \mathcal{M}, s \models \phi \\ 0 & \textit{else} \end{cases}$$

Recall, that $L(z)$ is the activation set for automata states:

Definition 14 (Activation set). *For an arbitrary state z of Z we define*

$$L(z) := \{a \in \Sigma_\rho \big| \exists z' \in Z_{A_\rho}(\delta_{A_\rho}(z,a) = z')\}$$

i.e. $L(z)$ is the set of all elements from Σ_ρ that emanate from z.

Definition 15 (End condition of a program). *Let ρ be a program and A_ρ its corresponding deterministic program automaton. The end conditions of a program ρ are those suffixes of form $\Phi?;\epsilon$, where $\Phi = true$ is possible.*

$$Fin_z(\mathcal{A}) = \begin{cases} true & \textit{iff } z \in E \\ false & \textit{iff } z \notin E \wedge \forall a \in L(z) : (\delta(z,a) \notin E) \\ \Phi_1 \vee ... \vee \Phi_n & \textit{iff } z \notin E \wedge \forall i (\Phi_i?;\epsilon \in L(z)) \wedge (\delta(z, \Phi_i?;\epsilon) \in E) \end{cases}$$

Proof (Theorem 1).

We will prove theorem 1 by induction on the length of paths.

Induction start: $|\sigma| = |\sigma^\times| = 1$: Using the standard semantics of CSL (cf. [4]) we obtain:

$$Pr\{\sigma^\times \in Path_{(s,z_0)}^{\mathcal{M}^\times} \mid \exists t \in I(\mathcal{M}^\times, \sigma^\times @ t \models \chi_{\mathcal{M}^\times})\} =$$
$$\int_0^t \sum_{(s',z') \in S^\times} \mathbf{R}((s,z_0),(s',z')) \cdot e^{-\mathbf{E}((s,z_0))\cdot x} \cdot Ind(\mathcal{M}^\times, (s',z'), \chi_{\mathcal{M}^\times}) dx$$

As the length of the path is one, $Ind(\mathcal{M}^{\times}, (s', z'), \chi_{\mathcal{M}^{\times}})$ is either 1 or 0, i.e. $\chi_{\mathcal{M}^{\times}}$ either holds in (s', z') or does not.

For the original formula, the probability measure can be characterised as follows:

$$Pr\{\sigma \in Path_s^{\mathcal{M}} \mid \mathcal{M}, \sigma \models \Phi[\rho]^{[0,t]}\Psi\} =$$
$$\int_0^t \sum_{\{\Phi?;a|\Phi?;a \in L(z) \wedge \mathcal{M}, s \models \Phi\}} \sum_{s' \in S} \mathbf{R}_a(s, s') \cdot e^{-\mathbf{E}(s)\cdot x} \cdot Ind(\mathcal{M}, s', \Psi \wedge Fin_{z'}(\mathcal{A}))dx$$

Therefore we will now show that:

$$\int_0^t \sum_{\{\Phi?;a|\Phi?;a \in L(z) \wedge \mathcal{M}, s \models \Phi\}} \sum_{s' \in S} \mathbf{R}_a(s, s') \cdot e^{-\mathbf{E}(s)\cdot x} \cdot Ind(\mathcal{M}, s', \Psi \wedge Fin_{z'}(\mathcal{A}))dx =$$
$$\int_0^t \sum_{s' \in S} \sum_{\{\Phi?;a|\Phi?;a \in L(z) \wedge \mathcal{M}, s \models \Phi\}} \mathbf{R}_a(s, s') \cdot e^{-\mathbf{E}(s)\cdot x} \cdot Ind(\mathcal{M}, s', \Psi \wedge Fin_{z'}(\mathcal{A}))dx =$$
$$\int_0^t \sum_{(s', Z') \in S^*} \mathbf{R}((s, Z_0), (s', Z')) \cdot e^{-\mathbf{E}((s,Z))\cdot x} \cdot Ind(\mathcal{M}^{\times}, (s', Z'), \chi_{\mathcal{M}^{\times}})dx$$

The last equation holds, since by construction of $\mathcal{M}^{\times}$ we can conclude that:

$$\sum_{\{\Phi?;a|\Phi?;a \in L(z) \wedge \mathcal{M}, s \models \Phi\}} \mathbf{R}(s, s') = \mathbf{R}((s, z_0), (s', z'))$$

Therefore and by construction it holds that the two outer sums are equal. By construction of $\mathcal{M}^{\times}$ from $\mathcal{M}$ we conclude:

$$\mathbf{E}(s) = \mathbf{E}((s, z_0))$$

$Ind(\mathcal{M}^{\times}, (s', z'), \chi_{\mathcal{M}^{\times}}) = Ind(\mathcal{M}, s', \Psi \wedge Fin_{z'}(\mathcal{A}))$ by construction, as those states are labelled with $\chi_{\mathcal{M}^{\times}}$ in which $Fin_{z'}(\mathcal{A}))$ and Ψ hold and in $\mathcal{A}$ an accepting state has been reached.

Induction step: We assume that for paths of length n the assumption holds, now we consider paths $\sigma^{\times}$ resp. σ of lenght $n+1$:

Let $\sigma^{\times'}$ resp. σ' be paths of length n, where $\sigma^{\times'}$ is suffix of $\sigma^{\times}$ and σ' is suffix of σ, then

$$Pr\{\sigma^{\times} \in Path_{(s,Z_0)}^{\mathcal{M}^{\times}} \big| \mathcal{M}^{\times}, \sigma^{\times}@t \models \chi_{\mathcal{M}^{\times}}\} =$$
$$\int_0^t \sum_{(s', Z') \in S^{\times}} \mathbf{R}((s, Z_0), (s', Z')) \cdot e^{-\mathbf{E}((s,Z_0))\cdot x} \cdot$$
$$Pr\{\sigma^{\times'} \in Path_{(s,Z_0)}^{\mathcal{M}^{\times}} \big| \mathcal{M}^{\times}, \sigma^{\times'}@(t-x) \models \chi_{\mathcal{M}^{\times}}\}$$

Analogously:

$$Pr\{\sigma \in Path_{s \in S}^{\mathcal{M}} \big| \mathcal{M}, \sigma \models \Phi[\rho]^{[0,t]}\Psi\} =$$

$$\int_0^t \sum_{\{\Phi?;a|\Phi?;a\in L(z)\wedge\mathcal{M},s\models\Phi\}} \mathbf{R}_a(s,s')\cdot e^{-\mathbf{E}(s)\cdot x}\cdot Pr\{\sigma'\in Path^{\mathcal{M}}_{s\in S}\big|\mathcal{M},\sigma'\models\Phi[\rho']^{\leq t-x}\Psi\}$$

where ρ' is the suffix of ρ. Using **I.H.** and the induction start we conclude that the theorem holds, i.e.

$$Pr\{\sigma^{\times}\in Path^{\mathcal{M}^{\times}}_{(s,Z_0)}\big|\mathcal{M}^{\times},\sigma^{\times}@t\models\chi_{\mathcal{M}^{\times}}\}=Pr\{\sigma\in Path^{\mathcal{M}}_{s\in S}\big|\mathcal{M},\sigma\models\Phi[\rho]^{[0,t]}\Psi\}$$

$\boxtimes$

8 Conclusions

In this paper we have introduced a property-driven symbolic semantics for the stochastic process algebra $\mathcal{YAMPA}$. We have shown its usage of a property-driven semantics for model checking probabilistic SPDL path formulae reduces both time and memory requirements. These savings can be considerable, as shown for the Kanban system, where an overhead of several billion states could be avoided. The numerical algorithms for stochastic model checking have a time complexity at least linear in state space size, so that an enormous overall time gain can be expected.

Generally, when doing numerical analysis of CTMCs with a huge state space some caution is required. As reported in [5], the accuracy of the numerical analysis depends on many factors, e.g. state space ordering, the actual iterative solution method, etc. But it must be stressed, that this is a problem that applies to all approaches that rely on numerical analysis. In fact, the probability masses on both the model, generated using property-driven state space generation, and the model using the "traditional" approach are identical. The experiments we conducted, on both the reduced and non-reduced model did not yield any differences.

In the future we plan to combine this property-driven semantics with some notion of bisimulation reduction in order to obtain further state-space reductions and to investigate the possibilities to transfer the results from [2] to the stochastic case.

References

1. M. Ajmone Marsan, G. Balbo, and G. Conte. A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. *ACM Transactions on Computer Systems*, 2(2):93–122, May 1984.
2. A. Aziz, T. Shiple, V. Singhal, R. Brayton, and A. Sangiovanni-Vincentelli. Formula-dependent equivalence for compositional CTL model checking. *Form. Methods Syst. Des.*, 21(2):193–224, 2002.
3. Ch. Baier, L. Cloth, B. R. Haverkort, M. Kuntz, and M. Siegle. Model checking markov chains with actions and state labels. *IEEE Transactions on Software Engineering*, 33(4):209–224, 2007.

4. Ch. Baier, B. Haverkort, H. Hermanns, and J.P. Katoen. Model-Checking Algorithms for Continuous-Time Markov Chains. *IEEE Trans. Software Eng.*, 29(7):1–18, July 2003.
5. A. Bell. *Distributed Evaluation of Stochastic Petri Nets.* PhD thesis, RWTH Aachen, Fakultät für Mathematik, Informatik und Naturwissenschaften, 2003.
6. M. Bernardo and R. Gorrieri. A Tutorial on EMPA: A Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time. *Theoretical Computer Science*, 202:1–54, 1998.
7. T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. North-Holland, Amsterdam, 1989.
8. P. Buchholz. Markovian Process Algebra: Composition and Equivalence. In U. Herzog and M. Rettelbach, editors, *Proc. of the 2nd Workshop on Process Algebras and Performance Modelling*, pages 11–30, Regensberg/Erlangen, July 1994. Arbeitsberichte des IMMD, Universität Erlangen-Nürnberg.
9. G. Ciardo and M. Tilgner. On the use of Kronecker operators for the solution of generalized stochastic Petri nets. Technical Report 96-35, ICASE, 1996.
10. E.M. Clarke, E.A. Emerson, and A. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *10th ACM Annual Symp. on Principles of Programming Languages*, pages 117–126, 1983.
11. R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for symbolic model checking in CCS. *Distributed Computing*, 6(3):155–164, 1993.
12. M. Fischer and R. Ladner. Propositional dynamic logic of regular programs. *J. Comput. System Sci.*, 18:194–211, 1979.
13. M. Fujita, P. McGeer, and J.C.-Y. Yang. Multi-terminal Binary Decision Diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2/3):149–169, April/May 1997.
14. N. Götz. Stochastische Prozeßalgebren – Integration von funktionalem Entwurf und Leistungsbewertung Verteilter Systeme. Ph.D. thesis, Universität Erlangen–Nürnberg, 1994 (in German).
15. H. Hermanns, U. Herzog, and J.-P. Katoen. Process algebra for performance evaluation. *Theoretical Computer Science*, 274(1-2):43–87, 2002.
16. J. Hillston. *A Compositional Approach to Performance Modelling.* Cambridge University Press, 1996.
17. J.-P. Katoen, T. Kemna, I. Zapreev, and D. Jansen. Bisimulation minimisation mostly speeds up probabilistic model checking. In *Proc. 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, LNCS 4424, pages 76–92. Springer, 2007.
18. M. Kuntz. *Symbolic Semantics and Verification of Stochastic Process Algebras.* PhD thesis, Universität Erlangen-Nürnberg, Institut für Informatik 7, 2006.
19. M. Kuntz and M. Siegle. Deriving symbolic representations from stochastic process algebras. In *Process Algebra and Probabilistic Methods, Proc. PAPM-PROBMIV'02*, pages 188–206. Springer, LNCS 2399, 2002.
20. M. Kuntz, M. Siegle, and E. Werner. CASPA - A Tool for Symbolic Performance and Dependability Evaluation. In *Proceedings of EPEW'04 (FORTE co-located workshop*, pages 293 – 307. Springer, LNCS 3236, 2004.
21. W. H. Sanders and L. M. Malhis. Dependability evaluation using composed SAN-based reward models. *Journal of Parallel and Distributed Computing*, 15(3):238–254, 1992.

22. M. Siegle. Advances in model representation. In L. de Alfaro and S. Gilmore, editors, *Process Algebra and Probabilistic Methods, Joint Int. Workshop PAPM-PROBMIV 2001*, pages 1–22. Springer, LNCS 2165, September 2001.