# Rate Lifting for Stochastic Process Algebra by Transition Context Augmentation

AMIN SOLTANIEH, Department of Computer Science, Bundeswehr University Munich, Germany
MARKUS SIEGLE, Department of Computer Science, Bundeswehr University Munich, Germany

This paper presents an algorithm for determining the unknown rates in the sequential processes of a Stochastic Process Algebra (SPA) model, provided that the rates in the combined flat model are given. Such a rate lifting is useful for model reverse engineering and model repair. Technically, the algorithm works by solving systems of nonlinear equations and – if necessary – adjusting the model's synchronisation structure, without changing its transition system. The adjustments cause an augmentation of a transition's context and thus enable additional control over the transition rate. The complete pseudo-code of the rate lifting algorithm is included and discussed in the paper, and its practical usefulness is demonstrated by two case studies. The approach taken by the algorithm exploits some structural and behavioural properties of SPA systems, which are formulated here for the first time and could be very beneficial also in other contexts, such as compositional system verification.

**ACM Reference Format:**
Amin Soltanieh and Markus Siegle. 2024. Rate Lifting for Stochastic Process Algebra by Transition Context Augmentation. *ACM Trans. Model. Comput. Simul.* 1, 1 (April 2024), 30 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Stochastic Process Algebra (SPA) is a family of formalisms widely used in the area of quantitative modelling, especially for performance and dependability evaluation of concurrent systems. Typical members of this family are PEPA [11], TIPP [7], EMPA [3], CASPA [16], but also the reactive modules language of probabilistic verification tools such as PRISM [17] and STORM [6]. SPA models are frequently used in probabilistic model checking projects, where models need to satisfy formal requirements expressed, e.g. with the help of temporal logics such as CSL [1]. When a model violates a given requirement, model repair [2, 5, 21] seeks to improve the model in such a way that the requirement will eventually hold. Model repair can take various forms (see, for instance, the recent work on automatic abstract machine repair of B formal models [4]), but in connection with SPA performability[1] models, repair by transition rate adaptation is a major technique, aiming to speed up or slow down certain transitions in the model in order to positively influence probabilistic branchings and to control a system's timing behaviour [8, 29].

This paper presents a solution to the following problem: For a compositional SPA specification with known original transition rates in its components, let rate modification factors for (a subset of) the transition rates in its flat low-level model be given. The task is to find new transition rates for the components of the high-level SPA model, such that the resulting rates in the flat model will be modified as desired. An alternative formulation of the same problem is: Given a compositional SPA specification where the transition rates in its components are unknown but all transition

[1]Performability is a term originally coined by John Meyer, expressing the simultaneous consideration of performance and dependability issues [19].

Authors' addresses: Amin Soltanieh, Department of Computer Science, Bundeswehr University Munich, Germany, amin.soltanieh@unibw.de; Markus Siegle, Department of Computer Science, Bundeswehr University Munich, Germany, markus.siegle@unibw.de.

rates of the associated flat low-level transition system are known, the task is to find the unknown transition rates for the components of the high-level SPA model. The first formulation is from the perspective of model checking and model repair, whereas the second one pertains to systems reverse engineering (to be more specific, one could call it rate reverse engineering). We will refer to both variants of the problem as “rate lifting problem”.

An algorithm that solves the rate lifting problem for SPA models with only $n = 2$ components was presented in [27], the nonlinear equation system involved being studied in [28]. However, developing a rate lifting algorithm for a general number $n \geq 3$ of processes turns out to be a much bigger challenge, since – firstly – SPA models with $n$ components may have a much more complex synchronisation structure than for $n = 2$, and it is the synchronisation structure which plays an essential role during the execution of the algorithm. Secondly, components of SPA models often contain selfloops (intended to synchronise with other components, thereby enabling a check of a component’s state and controlling the overall rate of the resulting transition), and – related to this – the transition system underlying a compositional SPA model is actually a collapsed multi-transition system [7, 11]. These two facts have to be considered during the necessary deconstruction of a flat transition, and they strongly contribute to the complication of the problem. So, in this paper we develop a rate lifting algorithm for SPA systems consisting of $n$ components, where $n$ is arbitrary. The algorithm will assign (new) values to the components’ transition rates and – under certain circumstances – it will change the synchronisation structure of the SPA model. The latter means that the algorithm may add actions to certain synchronisation sets and in consequence it will insert additional selfloops at some specific component states, but it will do this in such a way that the set of reachable states and the set of transitions of the overall model are not changed, only the transition rates of the overall model are set/changed as desired. Such a change in the synchronisation structure realises a “transition context augmentation”, since the context of a transition, consisting of a well-defined set of participating sequential processes (see Sec. 2.3), is augmented by additional processes. Since the rate of a flat transition is a function of the rates in the participating processes, this enables additional control over the transition rate. For determining the unknown rates, the algorithm sets up and aims to solve (possibly multiple) systems of nonlinear (precisely: multilinear) equations. The actual equation solving is treated as a black box in this paper, outsourced to external tools such as Matlab [14], Mathematica [15] or Gurobi [13], since this is another involved topic, beyond the scope of the paper. It is quite easy to see that an arbitrary assignment of rates to the transitions of the low-level transition system may not always be realisable by suitable rates in the components, i.e. not every instance of the rate lifting problem has a solution. Therefore, naturally, the algorithm presented in this paper will not always succeed. However, it is guaranteed that the algorithm will find a solution, if such a solution exists (see Sec. 5).

We build our algorithm based on certain structural and behavioural properties of SPA systems, which are exploited in the course of the algorithm. As one example, for a given transition in one of the SPA components, it is necessary to identify the partners which must or may synchronise with it. As another example, for an individual flat transition, we must be able to determine exactly the set of participating sequential processes, including possible selfloops or even multiple combinations of selfloops (see Sec. 2.3 and Sec. 3). To the best of our knowledge, these fundamental properties of SPA have not previously been addressed in the literature, which is surprising, since they could be very valuable also in other contexts. For example, in compositional system verification, distinguishing between different types of neighbourhoods of processes or determining the participating set of a transition is the key to establishing dependence / independence relations between processes.

To summarize, the two most important contributions of this paper, which is a substantially extended version of the conference paper [24], are:

(1) The paper contributes to the theory of SPA by formulating new structural and behavioural properties of SPA systems and the underlying flat transition system. There is a particular focus on the role of selfloops and – related to this – “parallel” transitions (see Sec. 3).
(2) The paper presents the first complete rate lifting algorithm for general SPA systems with an arbitrary number $n$ of components, based on the aforementioned structural and behavioural properties. If the desired rates cannot be realised in the unmodified system, the algorithm follows the strategy of “transition context augmentation”, which means artificially extending the control over a transition’s rate without changing the qualitative behaviour of the system.

The remainder of this paper is structured as follows: Sec. 2 recapitulates the basics of SPA and provides the necessary definitions. The neighbourhood relations are introduced and their properties established. Then the flat transition system underlying an SPA system is inspected, leading to the important notions of Participating Set and Involved Set. Sec. 3 is devoted to the aspect of “parallel” transitions, which occur especially in connection with selfloops. The important helper algorithm for determining all relevant selfloop combinations contributing to a transition is presented. The explanation of our new rate lifting algorithm can be found in Sec. 4. After an informal overview, the problem of spurious transitions, which must be absolutely avoided when modifying the synchronisation structure of an SPA system, is addressed, and afterwards the pseudo code of the algorithm, as well as that of auxiliary algorithms, provided in the appendices, is explained in some detail. Sec. 5 shows the correctness of the algorithm and analyses its asymptotical complexity. In Sec. 6, two case studies are described, which show how the new rate lifting algorithm behaves in practice. Finally, Sec. 7 summarises the paper and lists some topics for future research.

## 2 STRUCTURAL AND BEHAVIOURAL PROPERTIES OF SPA

The class of Markovian Stochastic Process Algebra models considered in this paper is simple but still very general, as given by the following definition:

*Definition 2.1. (SPA language)* For a finite set of actions $Act$, let $a \in Act$ and $A \subseteq Act$. Let $\lambda \in \mathbb{R}^{>0}$ be a transition rate. An SPA system $Sys$ is a process of type $Comp$, constructed according to the following grammar:

$$\begin{aligned} Comp &:= (Comp \mid\mid_A Comp) \bigm| Seq \\ Seq &:= 0 \bigm| (a, \lambda); Seq \bigm| Seq + Seq \bigm| V \end{aligned}$$

$Seq$ stands for sequential processes, and $Comp$ for composed processes. $V$ stands for a process variable for a sequential process, which can be used to define cyclic behaviour (including selfloops). One could add a recursion operator, the special invisible action $\tau$, hiding, renaming of processes, and other features, but this is not essential for our purpose. The semantics is standard, i.e. the SPA specification is mapped to the underlying flat transition system, which is an action-labelled CTMC (see e.g. [7, 10, 11]). In Sec. 3.1 we will comment on the important fact that the semantic model, in general, is actually a collapsed multi-transition system. Like in other SPA formalisms, we assume multiway synchronisation (in contrast to the two-way synchronisation of CCS [20]). That is, in our setting, the synchronisation of two $a$-transitions yields another $a$-transition (with well-defined rate), which can then participate in further $a$-synchronisations, etc.. When two $a$-transitions synchronise (one from process $P$ and one from process $Q$), the resulting rate is assumed to be a function $f(.\,,.)$ of the two partner rates (such as their product, minimum, maximum, arithmetic mean, . . . ), and possibly of the two context processes $P$ and $Q$. Thus the semantic rule for process synchronisation can be written as

$$\frac{P \xrightarrow{a,\lambda} P' \ \wedge \ Q \xrightarrow{a,\mu} Q'}{P \ ||_A \ Q \xrightarrow{a,f(P,Q,\lambda,\mu)} P' \ ||_A \ Q'} \ (a \in A)$$

In the remainder of this paper, we will assume that $f(P, Q, \lambda, \mu) = \lambda \cdot \mu$, since multiplication of rates is a de facto standard for Markovian SPAs, as implemented, for example, by the tools PRISM and STORM. Multiplication has favourable algebraic properties, and in connection with SPA notions of equivalence, multiplication of rates implies congruence properties [10]. In addition, multiplication of rates is also a good choice from a modelling point of view: In many SPA models, multiplication of rates is used in such a way that one of the synchronisation partners sets the rate ($\lambda$, say) and the other partner(s) use(s) the rate 1 (neutral), or provide(s) an acceleration factor ($> 1$) or a slowdown factor ($< 1$). If the rate resulting from the synchronisation of two or more processes were defined by an operator different from the product of the participating rates, the equations constructed by our rate lifting algorithm (see Sec. 4) would have to be changed accordingly, but apart from this change, the lifting algorithm would still work in exactly the same way. Even for synchronisation policies where the resulting rate depends not only on the two synchronising transitions but on the two partner processes $P$ and $Q$ as a whole – such as PEPA's sychronisation policy involving the apparent rate of the partner processes [11] – our algorithms would also still work in the same way, but the equations would become more complex in order to reflect that dependency.

It is important to mention that SPA processes may contain selfloops which will play an important role in this paper. For example, process $P = (a, \lambda).Q + (b, \mu).P$ may either move to $Q$ with action $a$ (at rate $\lambda$) or perform a selfloop with action $b$ (at rate $\mu$). Selfloops are often used as a valuable modelling feature to control the context of a transition, such as testing an enabling condition for a transition in another, synchronised process, or to control the rate of a synchronised transition. The latter feature is exploited by our rate lifting algorithm in Sec. 4, by inserting artificial selfloops in specific places, in order to realise the desired rates. In this paper, a selfloop in a single sequential process will be called an atomic selfloop, whereas the synchronisation of two or more selfloops from different sequential processes will be called a combined selfloop.

An SPA system (as defined in Def. 2.1) corresponds to a process tree whose internal nodes are labelled by the parallel composition operator, each one parametrized by a set of synchronising actions ($||_A$, with $A \subseteq Act$), and whose leaves are sequential processes of type $Seq$. For a specific action $a \in Act$, we write $||_a$ as an abbreviation to express that $a$ belongs to the set of synchronising actions, and $||_{\neg a}$ that it does not.

*Definition 2.2.* Let $Sys$ be a given SPA system.

(a) The set of all sequential processes within $Sys$ is denoted as $seqproc(Sys)$ (i.e. the set of all leaves of the process tree of $Sys$).

(b) The set of all (sequential or composed) processes within $Sys$ is denoted as $proc(Sys)$ (i.e. the set of all nodes of the process tree of $Sys$).

It holds that $seqproc(Sys) \subseteq proc(Sys)$, since the leaves of a tree form a subset of all the nodes of a tree.

Let us denote all actions occurring in the syntactical specification of a sequential process $P \in seqproc(Sys)$ as $Act(P)$. We can extend this definition to an arbitrary process $X \in proc(Sys)$ by writing $Act(X) = \bigcup Act(P_i)$, where the union is over those sequential processes $P_i$ that are in the subtree of $X$. Note that, throughout the paper, we will use symbols like $X, Y, \ldots$ to denote both, a (sequential or composed) process, and the associated node in the process tree with its subtree. For a sequential process $P$, the fact that $a \in Act(P)$ means that $P$ (considered in isolation) can actually at some point in

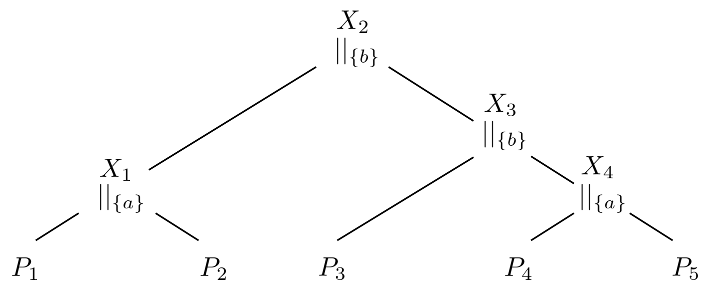

Fig. 1. $(P_1 \ ||_{\{a\}} \ P_2) \ ||_{\{b\}} \ (P_3 \ ||_{\{b\}} \ (P_4 \ ||_{\{a\}} \ P_5))$

its dynamic behaviour perform action $a$. However, for a process $X \in proc(Sys) \setminus seqproc(Sys)$, the fact that $a \in Act(X)$ does not necessarily mean that $X$ can actually perform action $a$. As an example, think of $X = P \ ||_a \ Q$, where $a \in Act(P)$ but $a \notin Act(Q)$. As another example, think of the same $X$ where $a \in Act(P)$ and $a \in Act(Q)$ but no combined state is reachable in which both $P$ and $Q$ can perform action $a$. Therefore, we define $Act_{perf}(X) \subseteq Act(X)$ to be those actions that $X$ (considered as a process in isolation) can actually perform at some point in its dynamic behaviour. Formally:

$$a \in Act_{perf}(X) \ \text{ iff } \ \exists \text{ state } s \text{ reachable from the initial state of } X : \exists \lambda : \exists s' : s \xrightarrow{a,\lambda} s'$$

While $Act(X)$ is a purely syntactical concept, $Act_{perf}(X)$ is a behavioural concept.

Given an SPA system $Sys$ and two proceses $X, Y \in proc(Sys)$, we say that $X$ and $Y$ are disjoint if and only if they do not share any part of the process tree of $Sys$. In other words: Neither is $X$ included in the subtree rooted at $Y$, nor vice versa. Inside the disjoint processes $X$ and/or $Y$, different actions (from $Act(X)$ and $Act(Y)$) may take place, among them the specific action $a$, say. Synchronisation on action $a$ between $X$ and $Y$ is possible if and only if the root of the smallest subtree containing both $X$ and $Y$ is of type $||_a$. Maximal $||_a$-rooted subtrees are called $a$-scopes, as formalized in the following definition.

*Definition 2.3.* Let $a \in Act$. An $a$-scope within an SPA system $Sys$ is a subtree rooted at a node of type $||_a$, provided that on all nodes on the path from that node to the root of $Sys$ there is no further synchronisation on action $a$ (i.e. all nodes on that path, including the root, are of type $||_{\neg a}$).
Furthermore, as a special case, if $P \in seqproc(Sys)$ and there is no $a$-synchronisation on the path from $P$ to the root of the process tree of $Sys$, we say that $P$ by itself is an $a$-scope.

For example, in the system shown in Figure 1, subtrees rooted at $X_1$ and $X_4$ are $a$-scopes, and sequential process $P_3$ is also an $a$-scope. The only $b$-scope of this system is at the root of the system, i.e. $X_2$.

Note that, according to this definition, $a$-scopes are always maximal, i.e. an $a$-scope can never be a proper subset of another $a$-scope. Clearly, if the root node of $Sys$ requires synchronisation on action $a$, then the whole $Sys$ is a single $a$-scope. Synchronisation via action $a$ is impossible between two distinct $a$-scopes. But even within a single $a$-scope, not all processes can / need to synchronise on action $a$, which motivates the neighbourhood relations in the next section.

## 2.1 Neighbourhood relations

The following definition answers the question (from the perspective of a sequential process $P$) which processes cannot / must / may synchronise with an $a$-transition in process $P$.

*Definition 2.4.* For $a \in Act$, consider the $a$-transitions within process $P \in seqproc(Sys)$. Let $X \in proc(Sys)$ be such that $P$ and $X$ are disjoint (i.e. that $P$ is not part of $X$). Let $r$ be the root of the smallest subtree that contains both $P$ and $X$.

(a) $X \in N_{cannot}(Sys, P, a)$ iff $r$ is of type $||_{\neg a}$.

(b) $X \in N_{must}(Sys, P, a)$ iff $r$ is of type $||_a$ and on the path from $r$ to $X$ all nodes are of type $||_a$, where "path" means all nodes strictly between $r$ and the root of $X$.

(c) $X \in N_{may}(Sys, P, a)$ iff $r$ is of type $||_a$ but on the path from $r$ to $X$ there exists a node of type $||_{\neg a}$.

For example, referring again to the system shown in Figure 1, $X_3 \in N_{cannot}(Sys, P_1, a)$ because $X_2$ is of type $||_{\neg a}$. $X_4 \in N_{must}(Sys, P_3, b)$ because $X_3$ is of type $||_b$. $P_4 \in N_{may}(Sys, P_3, b)$ because $X_3$ is of type $||_b$ but on the path from $P_4$ to $X_3$ there is a node (namely $X_4$) of type $||_{\neg b}$.

Remark 1. *In the definition, $P$ denotes a sequential process, whereas $X$ denotes any kind of process (sequential or composed). Notice that for a process $X$ it is possible that $X \in N_{may}(Sys, P, a)$ or $X \in N_{must}(Sys, P, a)$ even if $a \notin Act_{perf}(X)$ (or even $a \notin Act(X)$), which of course means that $X$ will never be able to synchronise on action $a$. This is considered below in the definition of the set $IS_r$ (cf. Def. 2.8). Related to this observation, note further that a process $X \in N_{may}(Sys, P, a)$ could actually be forced to synchronise with $P$ on action $a$ (i.e. it could be that $X$ must synchronise with $P$ on $a$, even though $X \notin N_{must}(Sys, P, a)$). For example, if $Sys = P\ ||_a\ (Q\ ||_{\neg a}\ R)$ then $Q \in N_{may}(Sys, P, a)$ and $R \in N_{may}(Sys, P, a)$, but if $a \notin Act(Q)$ then $P$ always needs $R$ as a synchronisation partner on action $a$.*

We proceed with some useful properties related to the neighbourhood relations:

Lemma 2.5. (a) *The neighbourhood $N_{cannot}(Sys, P, a)$ is disjoint from $N_{may}(Sys, P, a)$ and $N_{must}(Sys, P, a)$.*

(b) *Every $X \in N_{may}(Sys, P, a)$ is a subtree of some $Y \in N_{must}(Sys, P, a)$.*

Proof. Part (a) follows directly from the definition. Part (b): For given $X$, one such node $Y$ is the node directly below $r$ on the path from $r$ (as defined in Def. 2.4) to $X$. □

For a given sequential process $P$ and action $a$, the complete set $N_{must}(Sys, P, a)$ can be determined algorithmically as follows: When moving up from $P$ to the root of $Sys$, on every $||_a$-node the other subtree is in $N_{must}(Sys, P, a)$. Furthermore, when moving down recursively in such a subtree, as long as one moves along $||_a$-nodes, both subtrees are also in $N_{must}(Sys, P, a)$. Given a system with $n$ sequential processes (whose process tree has $2n - 1$ nodes), calculating $N_{must}(Sys, P, a)$ for a single process $P$ thus takes $O(n)$ steps. In a similar way, one can also describe algorithms that determine the complete set $N_{may}(Sys, P, a)$ or the complete set $N_{cannot}(Sys, P, a)$.

## 2.2 Symmetry and Transitivity of Neighbourhood Relations

The following symmetry and transitivity issues are not directly needed in our rate lifting algorithm, but they could be very useful in other applications.

Lemma 2.6. *For an SPA system $Sys$, let $P, Q \in seqproc(Sys)$.*

(a) *The $N_{cannot}$ relationship is symmetric, which means that $Q \in N_{cannot}(Sys, P, a)$ implies that $P \in N_{cannot}(Sys, Q, a)$.*

(b) *In contrast, the $N_{must}$ relationship is not symmetric, since $Q \in N_{must}(Sys, P, a)$ implies either that $P \in N_{must}(Sys, Q, a)$ or that $P \in N_{may}(Sys, Q, a)$.*

(c) *Similarly, the $N_{may}$ relationship is also not symmetric, since $Q \in N_{may}(Sys, P, a)$ implies either that $P \in N_{may}(Sys, Q, a)$ or that $P \in N_{must}(Sys, Q, a)$.*

Proof. Part (a): From $Q \in N_{cannot}(Sys, P, a)$ we know that $r$ (the root of the smallest subtree that contains both $P$ and $Q$) is of type $||_{\neg a}$. If we interchange the roles of $P$ and $Q$, the root of the relevant subtree is still the same $r$, so the condition for cannot-neighbourhood is also satisfied in the other direction. Part (b): From $Q \in N_{must}(Sys, P, a)$ we know that $r$ is of type $||_a$, so for sure $P \notin N_{cannot}(Sys, Q, a)$. That means that either $P \in N_{must}(Sys, Q, a)$ or $P \in N_{may}(Sys, Q, a)$. Both of these cases are possible, as can be easily shown by example: If $Sys = P \; ||_a \; Q$, then $Q \in N_{must}(Sys, P, a)$ and $P \in N_{must}(Sys, Q, a)$. However, if $Sys = (P \; ||_{\neg a} \; R) \; ||_a \; Q$ then $Q \in N_{must}(Sys, P, a)$ but $P \in N_{may}(Sys, Q, a)$, because an $a$-transition of $Q$ could synchronise with $R$ instead of $P$. Part (c) can be shown in a similar way as Part (b). □

Lemma 2.7. *For an SPA system Sys, let $P, Q, R \in seqproc(Sys)$.*

(a) *The $N_{must}$ relationship is transitive, which means that*

$$Q \in N_{must}(Sys, P, a) \;\wedge\; R \in N_{must}(Sys, Q, a) \;\Rightarrow\; R \in N_{must}(Sys, P, a)$$

(b) *In contrast, the $N_{may}$ and $N_{cannot}$ relationships are not transitive.*

Proof. Starting with Part (b), this can be easily shown by counterexamples: For $N_{cannot}$ one counterexample is $Sys = ((P \; ||_a \; R) \; ||_{\neg a} \; Q)$. For $N_{may}$ one counterexample is $Sys = ((P \; ||_{\neg a} \; R) \; ||_a \; (Q \; ||_{\neg a} \; S))$. For Part (a), a formal proof arguing along the structure of the process tree can be found in [26]. However, it is also intuitively clear that if $Q$ is forced to go along with $P$ and if $R$ is forced to go along with $Q$ in any $a$-transition, then indirectly $R$ is also forced to go along with $P$. □

## 2.3 Moving Set and Participating Set

Having discussed the neighbourhoods of processes, which are related to the structure of SPA systems, we now focus on the dynamic behaviour of SPA systems. Given a system $Sys$ constructed from $n$ sequential processes $P_1, \ldots, P_n$, its global state is a vector $(s_1, \ldots, s_n)$ where $s_i$ is the state of $P_i$ (a state of a sequential SPA process $P_i$ is just a derivative of the process, i.e. any SPA process reachable from $P_i$) . We follow the convention that the ordering of processes in the state vector is given by the in-order (LNR) traversal of the process tree of $Sys$. A transition $t$ in the flat transition system of $Sys$ is given by

$$t = ((s_1, \ldots, s_n) \xrightarrow{a, \lambda_s} (s'_1, \ldots, s'_n))$$

where for at least one $k \in \{1, \ldots, n\}$ we have $s_k \neq s'_k$ and where the transition rate $rate(t) = \lambda_s$ is a function of the rates of the transitions of the participating processes. The assumption that in each such transition at least one of the $n$ processes must move makes sense, since we are considering here transitions of the overall system $Sys$, where a global selfloop would be meaningless for our work and therefore could be ignored. For such a transition $t$ we introduce the following notation:

$$\begin{aligned} action(t) &= a & rate(t) &= \lambda_s \\ source(t) &= (s_1, \ldots, s_n) & target(t) &= (s'_1, \ldots, s'_n) \\ source_i(t) &= s_i & target_i(t) &= s'_i \end{aligned}$$

But which are actually the participating processes in the above transition $t$? For an $a$-transition $t$ as above, we define the Moving Set $MS(t)$ as the set of those sequential processes whose state changes, i.e. $MS(t) = \{P_k \mid s_k \neq s'_k\}$. The complement of the Moving Set is called the Stable Set $SS(t)$, i.e. $SS(t) = \{P_1, \ldots, P_n\} \setminus MS(t)$.

Since processes may contain selfloops and since synchronisation on selfloops is possible, the Participating Set $PS(t)$ of transition $t$ can also include processes which participate in $t$ in an invisible way by performing a selfloop. Therefore $PS(t)$ can be larger than $MS(t)$, i.e. in general we have $MS(t) \subseteq PS(t)$. Processes in $SS(t)$ which *must* synchronise on $a$ with one of the elements of $MS(t)$ must have an $a$-selfloop at their current state and must belong to $PS(t)$. Furthermore, processes in $SS(t)$ which *may* synchronise on $a$ with one of the elements of $MS(t)$ and have an $a$-selfloop at their current state may also belong to $PS(t)$, provided that they are not in the $N_{cannot}$-neighbourhood of one of the processes of $MS(t)$. Altogether we get:

$$\begin{aligned} PS(t) \quad = \quad & MS(t) \cup \\ & \Big\{ P_i \in SS(t) \mid \\ & \Big( \exists P_j \in MS(t) : \\ & \big(P_i \in N_{must}(Sys, P_j, a) \\ & \vee \big(P_i \in N_{may}(Sys, P_j, a) \wedge (\text{selfloop } s_i \xrightarrow{a,\lambda_i} s_i \text{ exists and is enabled in } source(t))\big)\big)\Big) \\ & \wedge \big(\nexists P_j \in MS(t) : P_i \in N_{cannot}(Sys, P_j, a)\big) \Big\} \end{aligned} \tag{1}$$

The condition "selfloop … is enabled in $source(t)$" means that the selfloop in $P_i$ can actually take place in the source state of transition $t$, i.e. it is not blocked by any lacking synchronisation partner(s). Note that for the case $P_i \in N_{must}(Sys, P_j, a)$ there obviously exists a selfloop in process $P_i$, but this existence is implicit, so we do not have to write it down. In Sec. 2.4, a procedure for practically calculating $PS(t)$ is given.

Using an example we show why the definition of $PS(t)$ needs to be so complicated, in particular why being in the *may* neighbourhood of a moving component and having a selfloop is not enough to become a participating component. For the system shown in Figure 2, we wish to find $PS(t)$ where $t = ((s_1, s_2, s_3, s_4, s_5) \xrightarrow{a,\,\cdot} (s_1', s_2, s_3, s_4, s_5))$. $P_1$ is the only moving component, and assume that there are $a$-selfloops in state $s_2$ of $P_2$ and also in state $s_3$ of $P_3$, but that there are no $a$-selfloops in state $s_4$ of $P_4$ and in state $s_5$ of $P_5$. $P_2 \in N_{may}(Sys, P_1, a)$ is in $PS(t)$, since its selfloop can take place without hindrance, whereas $P_3 \in N_{may}(Sys, P_1, a)$ is not included in $PS(t)$, since its selfloop, although it exists, is not enabled in the source state of transition $t$ (it would need $P_4$ or $P_5$ as a synchronisation partner).

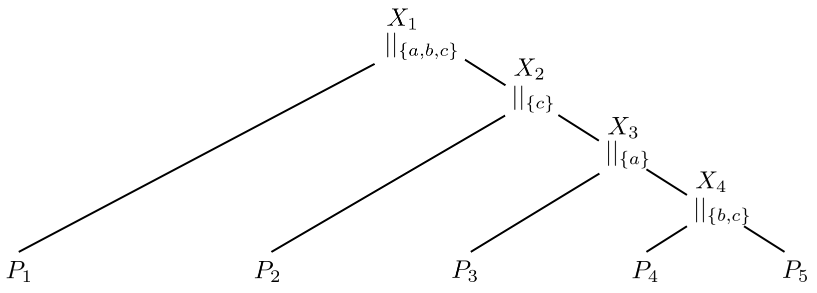

Fig. 2. $P_1 ||_{\{a,b,c\}} (P_2 ||_{\{c\}} (P_3 ||_{\{a\}} (P_4 ||_{\{b,c\}} P_5)))$

## 2.4 Calculating the Participating Set

In Eq. 1 quite a complicated closed-form expression for the Participating Set of a transition was given. In practice, the Participating Set $PS(t)$ for a given transition $t$ in an SPA system, with $action(t) = a$, can be obtained by the following procedure: We first calculate a set of candidate processes $PS_{cand}$:

$$\begin{aligned} PS_{cand}(t) = \quad & \{P_i \in SS(t) \mid (P_i \text{ has a selfloop } s_i \xrightarrow{a,\lambda_i} s_i) \\ & \wedge (\exists P_j \in MS(t) : (P_i \in N_{may}(Sys, P_j, a))) \\ & \wedge (\nexists P_j \in MS(t) : (P_i \in N_{cannot}(Sys, P_j, a)))\} \end{aligned}$$

Then we determine the set $PS_{may} \subseteq PS_{cand}$ as follows: For each $P_i \in PS_{cand}(t)$, let $r$ of type $\|_a$ be the root of the smallest subtree containing $P_i$ and at least one of the components of $MS(t)$. For each node $N$ of type $\|_a$ on the path from $P_i$ to $r$, excluding $r$, let $X_N$ be the child process of node $N$ which does not contain $P_i$. If a selfloop $(\vec{s}_{X_N} \xrightarrow{a,\,\cdot} \vec{s}_{X_N})$ exists in process $X_N$ then $P_i \in PS_{may}(t)$[2]. Overall we get:

$$PS(t) = MS(t) \;\cup\; PS_{may}(t) \;\cup\; \{P_i \in SS(t) \mid \exists P_j \in MS(t) : P_i \in N_{must}(Sys, P_j, a)\}$$

Returning to the example from Sec. 2.3 (Figure 2): For transition $t = ((s_1, s_2, s_3, s_4, s_5) \xrightarrow{a,\,\cdot} (s_1', s_2, s_3, s_4, s_5))$, we assumed that there are selfloops in state $s_2$ in $P_2$ and also in state $s_3$ in $P_3$, so both $P_2$ and $P_3$ are in $PS_{cand}(t)$. Since on the path from $P_2$ to $r$ $(= X_1)$ there is no node of type $||_a$, nothing else needs to be checked for $P_2$, so $P_2$ is in $PS_{may}$. For $P_3$, we need to check whether the other child of $X_3$ (which is $X_4$) can peform an $a$-selfloop at the current state $\vec{s}_{X_4} = (s_4, s_5)$. But since this is not the case, $P_3$ is not in $PS_{may}$ and therefore not in $PS(t)$.

## 2.5 Involved Set

In addition to the Participating Set $PS(t)$ of a transition $t$, we also need to define the Involved Set $IS(t)$ which can be larger than $PS(t)$, since it also contains those processes which may synchronise on action $a$ with one of the processes in $PS(t)$ (in another transition $t'$), and so on, inductively. Formally:

*Definition 2.8.* For a transition $t$ with $action(t) = a$ we define

(i) The Involved Set

$$IS(t) \quad = \quad PS(t) \cup \Big\{P_k \in seqproc(Sys) \mid \exists P_j \in IS(t) : \big(P_k \in N_{may}(Sys, P_j, a)\big)\Big\}$$

(ii) The restricted Involved Set $IS_r(t) = \{P \in IS(t) \mid action(t) \in Act(P)\}$.

So $IS(t)$ represents the transitive closure of the $N_{may}$-neighbourhood of one of the participating processes. For that reason, after the existential quantifier in part (i) we have to write $P_j \in IS(t)$ instead of only $P_j \in PS(t)$. The definition of the restricted Involved Set $IS_r(t)$ is motivated by the observation in Remark 1. The idea is to omit those sequential processes $P$ from $IS(t)$ where $a \notin Act(P)$, since they will never actually synchronise on action $a$ with any other process.

In some cases $IS(t) = PS(t)$, but it can be easily shown by example that $IS(t)$ may be a strict superset of $PS(t)$. As an example, consider the system

$$Sys = (P_1 \; ||_{\neg a} \; P_2) \; ||_a \; (P_3 \; ||_a \; P_4)$$

and the transition $t = ((s_1, s_2, s_3, s_4) \xrightarrow{a,\lambda_s} (s_1', s_2, s_3', s_4))$. The Moving Set is $MS(t) = \{P_1, P_3\}$, and there is obviously a selfloop in $P_4$ of the form $s_4 \xrightarrow{a,\lambda_4} s_4$ (otherwise $P_3$ would not be able to perform the $a$-transition $s_3 \xrightarrow{a,\lambda_3} s_3'$), so the Participating Set is $PS(t) = \{P_1, P_3, P_4\}$. However, $P_2$ is also (indirectly) involved since it is possible that in some other transition $P_3$ (and $P_4$) will synchronise on action $a$ with $P_2$. More concretely: The transitions $s_3 \xrightarrow{a,\lambda_3} s_3'$ (in

[2]The notation $\vec{s}_{X_N}$ denotes a vector representing the states of the sequential components in process $X_N$, i.e. it is the projection of the overall state vector $\vec{s} = (s_1, \ldots, s_n)$ to $X_N$.

$P_3$) and $s_4 \xrightarrow{a,\lambda_4} s_4$ (in $P_4$) may synchronise with $s_2' \xrightarrow{a,\lambda_2'} s_2''$ (in $P_2$) (for some states $s_2'$ and $s_2''$ of $P_2$). Therefore we get $IS(t) = \{P_1, P_2, P_3, P_4\}$. This will be important for our rate lifting algorithm (Sec. 4), since if we didn't take the involvement of $P_2$ into account, we might (while processing transition $t$) change some rates in $P_3$ and/or $P_4$ which would have side effects on other transitions. This means that in our rate lifting algorithm we will have to set up a system of equations involving all four processes.

The following lemma establishes the connection between a transition's Involved Set $IS(t)$ (a behavioural concept) and the $a$-scope from Def. 2.3, which is a structural concept.

Lemma 2.9. *For a transition $t$ of the SPA system $Sys$, with $action(t) = a$, let $r$ be the root of the smallest tree containing all processes of $IS(t)$.*

(i) *Then $r$ is a node of type $||_a$.*
(ii) *There is no other node of type $||_a$ "above" $r$ (i.e. on the path from $r$ to the root of $Sys$).*
(iii) *The Involved Set $IS(t)$ is exactly the set of all sequential processes in the subtree rooted at $r$.*
(iv) *The Involved Set $IS(t)$ is exactly the set of sequential processes in the $a$-scope rooted at $r$. So, in a sense, the Involved Set and the $a$-scope are equal.*

Proof. (i) Assume that $r$ was of type $||_{\neg a}$. Then no process $P_l \in IS(t)$ in the left subtree of $r$ could synchronise (on action $a$) with any process $P_r \in IS(t)$ in the right subtree of $r$, which contradicts the fact that the set $IS(t)$ contains processes in both subtrees of $r$.
(ii) Furthermore, assume that there is another node $r_2$ of type $||_a$ on the path from $r$ to the root of $Sys$. Then any $a$ transition occurring in one of the processes of $IS(t)$ would have to synchronise with some process in the other subtree of $r_2$, which means that the subtree rooted at $r$ does actually not contain all processes of $IS(t)$, which is a contradiction.
(iii) We only need to show that each sequential process in the subtree rooted at $r$ is in $IS(t)$. Assume that there is a sequential process $P_{not}$ in the left subtree of the tree rooted at $r$ such that $P_{not} \notin IS(t)$. We know that there exists a sequential process $P_r$ in the right subtree of the tree rooted at $r$ such that $P_r \in IS(t)$. Then, since according to (i) $r$ is of type $||_a$, either $P_{not} \in N_{may}(Sys, P_r, a)$ or $P_{not} \in N_{must}(Sys, P_r, a)$. But from this it follows that $P_{not}$ would have to be in $IS(t)$, which is a contradiction. A symmetric argument holds if we assume that there is a sequential process $P_{not}$ in the right subtree of the tree rooted at $r$.
(iv) This is an immediate consequence of (i) - (iii). □

Lemma 2.10. *For two transitions $t_1$ and $t_2$ with $action(t_1) = action(t_2)$, if $IS(t_1) \cap IS(t_2) \neq \emptyset$ then $IS(t_1) = IS(t_2)$.*

Proof. This follows directly from the closure property of the $IS$ definition. □

## 3 "PARALLEL" TRANSITIONS AND RELEVANT SELFLOOP COMBINATIONS

### 3.1 Multi Transition System

It is well known that the semantic model underlying an SPA specification is actually a *multi*-transition system [7, 11]. This is usually collapsed to an ordinary transition system by adding up the rates of "parallel" transitions, i.e. transitions which have the same source state, the same target state and the same action label. Thus a transition within the flat transition system of $Sys$ may be the aggregation of more than one transition. As an example, consider the system $Sys = P\ ||_a\ (Q\ ||_{\neg a}\ R)$ and the transition $t = ((s_1, s_2, s_3) \xrightarrow{a,\lambda_s} (s_1', s_2, s_3))$. The Moving Set is $MS(t) = \{P\}$, but the Participating Set must be larger. Assume that $Q$ has a selfloop $s_2 \xrightarrow{a,\lambda_2} s_2$ and that $R$ has a selfloop $s_3 \xrightarrow{a,\lambda_3} s_3$. Since

$Q$ and $R$ do not synchronise on $a$, only one of those two selfloops synchronises with $s_1 \xrightarrow{a,\lambda_1} s_1'$ at a time, but both selfloops may synchronise with the $a$-transition in $P$. This yields the two "parallel" transitions

$$((s_1, s_2, s_3) \xrightarrow{a,\lambda_{12}} (s_1', s_2, s_3)) \quad \text{and} \quad ((s_1, s_2, s_3) \xrightarrow{a,\lambda_{13}} (s_1', s_2, s_3))$$

(where $\lambda_{12}$ is a function of $\lambda_1$ and $\lambda_2$, and $\lambda_{13}$ is a function of $\lambda_1$ and $\lambda_3$), but they are not visible as separate transitions in the low-level transition system. Rather, they are aggregated to the single transition $((s_1, s_2, s_3) \xrightarrow{a,\lambda_{12}+\lambda_{13}} (s_1', s_2, s_3))$, so $\lambda_s = \lambda_{12} + \lambda_{13}$. As an anticipation of Eq. 4 in Sec. 4, let us mention that in this situation our rate lifting algorithm would create the equation

$$x^{(P)}_{s_1 s_1'} x^{(Q)}_{s_2 s_2} + x^{(P)}_{s_1 s_1'} x^{(R)}_{s_3 s_3} = \lambda_s \cdot f \tag{2}$$

where a variable of the form $x^{(P_i)}_{s,s'}$ denotes the unknown rate from state $s$ to state $s'$ in sequential process $P_i$, $f$ is the desired rate multiplication factor, and (as discussed in Sec. 2) synchronisation is realised by multiplication of rates.

### 3.2 Calculating Relevant Selfloop Combinations

In the simple (and most common) case that none of the sequential processes in the SPA specification of $Sys$ has any selfloops (and also no "parallel" transitions), we know that any transition of the flat transition system has only one single semantic derivation. In consequence, for the considered flat transition $t$ it then holds that $PS(t) = MS(t)$. However, as discussed above, in the general case the flat transition system underlying a compositional SPA specification is actually a multi-transition system which gets collapsed to an ordinary transition system by amalgamating "parallel" transitions. In order to cover this general case, in the lifting algorithm (see Sec. 4) we have to do the opposite: Instead of amalgamation, we need to deconstruct a flat transition into its constituents. I.e., given a flat transition (which was possibly amalgamated from parallel transitions), we need to find out the contributing transitions, in order to be able to construct the correct equation in line 52 of the algorithm (Eq. 4 in Sec. 4).

Consider the flat transition $t := ((s_1, \ldots, s_n) \xrightarrow{c,\gamma\cdot f} (s_1', \ldots, s_n'))$. We can determine its (non-empty) Moving Set $MS(t)$ and its Participating Set $PS(t)$, where we know that $MS(t) \subseteq PS(t)$. We are particularly interested in the processes from the set $(PS(t) \cap SS(t)) \setminus \bigcup_{P \in MS(t)} N_{must}(Sys, P, c)$, since these are exactly the processes that may (but not must) contribute to transition $t$. Certain combinations of these processes (which have selfloops, otherwise they wouldn't be in $PS(t)$) contribute to transition $t$. We call these combinations "relevant selfloop combinations (rslc)". Note that there are also selfloops in $\bigcup_{P \in MS(t)} N_{must}(Sys, P, c)$, but they are not part of rslc.

It remains to calculate rslc for transition $t$, still assuming that $action(t) = c$. For this purpose, we define a function $rslc(t)$ which returns a set of sets of sequential processes, each such set describes a relevant selfloop combination. In the process tree of $Sys$, let $r$ be the root node of the smallest subtree containing $PS(t)$. We know that $r$ is either a non-leaf node of type $||_c$ or a leaf (if $r$ were a non-leaf node of type $||_{\neg c}$, the Participating Set $PS(t)$ couldn't span both subtrees of $r$). Calling the recursive algorithm in Appendix B by the top-level call RSLC$(t, r)$ delivers all the relevant selfloop combinations. Note that $rslc(t)$ as called in the lifting algorithm has one argument (a transition) but the recursive function $RSLC(t, n)$ (see Appendix B) has two arguments (a transition and a node of the process tree, the latter needed for recursive descent through the process tree).

Example: Consider the SPA specification $Sys = ((P \;||_{\neg c}\; Q) \;||_c\; (R \;||_{\neg c}\; S)) \;||_c\; (T \;||_{\neg c}\; U)$ whose process tree is shown in Figure 3, and the transition $t := ((s_P, s_Q, s_R, s_S, s_T, s_U) \xrightarrow{c,\gamma\cdot f} (s_P', s_Q, s_R, s_S, s_T, s_U))$. Obviously, the Moving Set is $MS(t) = \{P\}$, and if we assume that there are $c$-selfloops in states $s_R, s_S, s_T$ and $s_U$ (in all of them!), the Participating

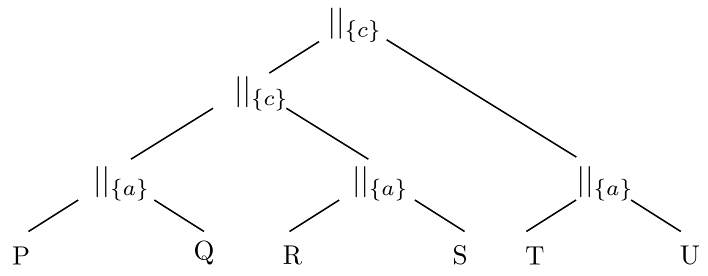

Fig. 3. $((P \,||_{\neg c}\, Q) \,||_c\, (R \,||_{\neg c}\, S)) \,||_c\, (T \,||_{\neg c}\, U)$

Set is $PS(t) = \{P, R, S, T, U\}$. So transition $t$ can be realised as any combination of a selfloop in $R$ or $S$ with a selfloop in $T$ or $U$, thus the algorithm will find the set of relevant selfloop combinations $\{\{R, T\}, \{R, U\}, \{S, T\}, \{S, U\}\}$. Anticipating once again Eq. 4 from Sec. 4, this set of relevant selfloop combinations would lead to the desired equation

$$x^{(P)}_{s_P s'_P} x^{(R)}_{s_R s_R} x^{(T)}_{s_T s_T} + x^{(P)}_{s_P s'_P} x^{(R)}_{s_R s_R} x^{(U)}_{s_U s_U} + x^{(P)}_{s_P s'_P} x^{(S)}_{s_S s_S} x^{(T)}_{s_T s_T} + x^{(P)}_{s_P s'_P} x^{(S)}_{s_S s_S} x^{(U)}_{s_U s_U} = \gamma \cdot f \tag{3}$$

where, again, a variable of the form $x^{(P_i)}_{s,s'}$ denotes the unknown rate from state $s$ to state $s'$ in process $P_i$. This equation reflects the fact that process $P$, moving from $s_P$ to $s'_P$, is participating, always together with exactly one of the four relevant selfloop combinations. Alternatively, if we assumed that the Participating Set was smaller, say $PS(t) = \{P, R, S, T\}$ (i.e. if there were no $c$-selfloop at $s_U$), then the algorithm would find a smaller set of relevant selfloop combinations, namely $\{\{R, T\}, \{S, T\}\}$, leading to the simpler equation $x^{(P)}_{s_P s'_P} x^{(R)}_{s_R s_R} x^{(T)}_{s_T s_T} + x^{(P)}_{s_P s'_P} x^{(S)}_{s_S s_S} x^{(T)}_{s_T s_T} = \gamma \cdot f$.

## 4 RATE LIFTING ALGORITHM

Our rate lifting algorithm processes the transitions whose rates are to be modified in a one by one fashion. It is, however, not strictly one by one, since in many situations a whole set of "related" transitions is taken into account together with the currently processed transition, where "related" means that the transitions have the same action label and take place within the same scope. The algorithm consists of four parts named A, B, C and D as shown in Figure 4. In Part A, for a transition whose Involved Set consists of only one single sequential process, the algorithm first tries to change its rate by modifying the rate locally in exactly this sequential process, which is referred to as "local repair". Local repair will fail, however, if two flat "sibling" transitions which both originate from the same local transition have different modification factors. In Part B, which is the starting point for transitions whose Involved Set contains at least two processes, the algorithm creates a system of nonlinear equations and tries to solve it. This system of equations covers all transitions with the same action label and the same Involved Set, i.e. all these transitions are dealt with simultaneously in one system of equations. The basic idea behind the system of equations is to consider all involved local rates as variables whose values are to be determined. Part C, entered upon failure of Part B, is the first part where the system specification is modified by augmenting some synchronisation sets and inserting selfloops, all within the current $c$-scope. These modifications are carried out in such a way that the global transition system is not changed. Again, like in part B, the algorithm creates a set of nonlinear equations (but now the system of equations is larger, involving more unknown rates, since the model has been modified) and tries to solve it. If the previous steps have failed, Part D tries to expand the scope, by modifiying the system in a larger scope than the current Involved Set. This means that the Involved Set is artificially augmented by adding action $c$ to the synchronisation set at a higher node. Again, a

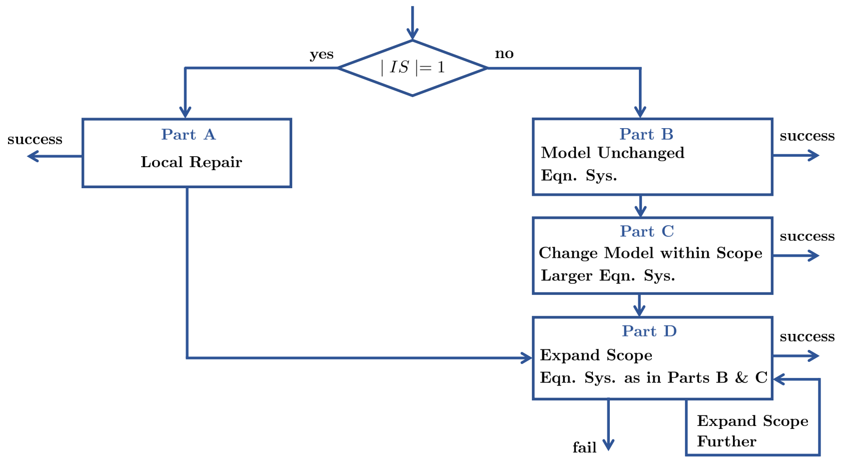

Fig. 4. Overview of the algorithm

similar but even larger system of equations of the same type is constructed. However, even this system of equations may not have a solution, in which case the desired rate lifting has turned out to be impossible.

## 4.1 Spurious Transitions

As we have seen, in certain situations the rate lifting algorithm needs to change the synchronisation structure of the given system, i.e. it will change an inner node of type $||_{\neg c}$ to a node of type $||_c$. Clearly, this needs to be done with great care, since such a step will – in general – change the behaviour of the system. Therefore the algorithm, before adding action $c$ to a synchronisation set, has to ensure that no spurious transitions will be generated. Spurious transitions (sp. tr.) are extra, superfluous transitions not present in the original system, and therefore incorrect. Furthermore, after action $c$ has been added to a synchronisation set, the algorithm also has to ensure that all transitions in the original system are still possible (it could easily happen that a previously existing transition now lacks a synchronisation partner in the newly synchronised system). For that purpose, the algorithm inserts selfloops into the sequential components, wherever necessary to preserve the behaviour of the original system. There are actually two types of spurious transitions:

(A) Superfluous transitions which appear when two previously $c$-non-synchronised components become synchronised over action $c$.

(B) Superfluous transitions which appear when a new $c$-selfloop is inserted into a sequential process which is in the $c$-must- or $c$-may-neighbourhood of another process.

The algorithm avoids both types of sp. tr.. It guarantees that even though the synchronisation structure of the system may be altered and artificial selfloops are inserted, the set of reachable states and the set of transitions remain the same.

## 4.2 Detailed Description of the Algorithm

Appendix A shows the pseudo code of the new rate lifting algorithm. The arguments of the algorithm are the SPA system $Sys$ and its flat transition system $T$ (a set of transitions), the set of transitions whose rate is to be modified $T_{mod} \subseteq T$ as well as a function $factor$ that returns, for each transition in $T_{mod}$, its modification factor (thus, this presentation of the algorithm addresses model repair rather than rate reverse engineering, cf. Sec. 1). For transitions not in $T_{mod}$, the modification factor is supposed to be 1.

In each iteration of the outer while-loop (lines 10 to 133), the algorithm picks one of the remaining transitions from $T_{mod}$, called $\hat{t}$ with action label called $c$, and processes it (possibly together with other transitions that have the same action label).

**(Part A) Local repair:** If the Involved Set of the currently processed transition $\hat{t}$ consists of only one single process, the algorithm tries to adjust the rate of exactly one transition in that process. This is done in lines 17–32 of the algorithm. However, this will only work if all "sibling" transitions where this process makes the same move have the same, common modification factor. That set of "sibling" transitions is denoted as $T_{c,\hat{s}_{i_1},\hat{s}'_{i_1}}$ in the algorithm, and the common modification factor is denoted as $f_{com}$.

**(Part B) System of equations for $T_c$:** If the Involved Set of the currently processed transition $\hat{t}$ consists of two or more processes, all transitions with the same Involved Set and the same action as $\hat{t}$ are processed together (lines 44–61). In the algorithm, this set of transitions is denoted $T_c$. For every transition $t \in T_c$, the algorithm determines its Participating Set $PS(t)$, calculates the relevant selfloop combinations (rslc) and from this information creates a nonlinear equation

$$\sum_{C \in rslc(t)} \prod_{P \in MS(t)} x^{(P)}_{s_P s'_P} \prod_{\substack{Q \in PS(t) \setminus MS(t) \\ \wedge\, \exists P \in MS(t):\, Q \in N_{must}(Sys,P,c)}} x^{(Q)}_{s_Q s_Q} \prod_{R \in C} x^{(R)}_{s_R s_R} = \gamma \cdot f \qquad (4)$$

where the $x$'s are the unknown rates of the participating processes (some of which are rates of selfloops, if such exist in the system). The superscript of variable $x^{(\cdot)}_{\cdot\cdot}$ identifies the sequential process, and the subscript denotes the source/target pair of states. The equation reflects the fact that the rates of all synchronising processes are multiplied, and that the total rate is obtained as the sum over all possible relevant selfloop combinations. As concrete examples for Eq. 4, Eqns. 2 and 3 were already shown in Sec. 3. Afterwards, this system of equations is solved, and if a solution exists, all $c$-transitions in the current $c$-scope have been successfully dealt with. We would like to point out that, if for some transition $t$ the Participating Set $PS(t)$ is equal to its Moving Set $MS(t)$, then the resulting equation has a much simpler form

$$x^{(P_1)}_{s_1 s'_1} \cdot x^{(P_2)}_{s_2 s'_2} \cdot \dots \cdot x^{(P_k)}_{s_k s'_k} = \gamma \cdot f$$

(assuming that $|MS(t)| = k$), since in this case, there are no selfloops involved, and therefore also no combinations of selfloops to be considered.

**(Part C) Expanding the context of $c$-transitions by synchronising with more processes and inserting artificial selfloops within the current $c$-scope:** If the system of equations constructed in part (B) for the set $T_c$ had no solution, it is the strategy of the algorithm to involve more processes (for the moment only from the current $c$-scope), since this opens up more opportunity for controlling the context of these $c$-transitions, and thereby controlling their rates. In this part of the algorithm (lines 66–100), $action(\hat{t}) = c$ is added to the synchronisation set at each node of type $||_{\neg c}$ in the current $c$-scope, except where this would lead to spurious transitions (of type A or type B). These tasks of the algorithm are outsourced to function TRYSYNC (called in line 77). In TRYSYNC (the details of which are elaborated on in Appendix C), checking for spurious transitions of type A is done by checking all source states of transitions in the current $T_c$, making sure that there are no concurrently enabled $c$-transitions in newly synchronised subprocesses. After adding action $c$ to some synchronisation sets, we also have to make sure that all transitions originally in $T_c$ can still occur, i.e. that they have not been disabled by the new synchronisations (because some newly necessary synchronisation partners would now be lacking). This is also done in function TRYSYNC, by inserting the necessary selfloops in those processes which are now newly synchronising on action $c$, provided that those new selfloops do not lead to the existence of spurious transitions (of type B). The steps just described guarantee that the modified system $Sys'$ has exactly the same set of transitions as the original system $Sys$ (qualitatively), but it remains to find the correct rates of all $c$-transitions in the involved processes. For this purpose, a similar (but larger) system of equations as in Part (B) is set up and solved.

**(Part D) Expanding the Involved Set by moving the current root upwards:** It is possible that the systems of equations constructed in Part (B) and thereafter in Part (C) both have no solution. In this case, the algorithm seeks to expand the current $c$-scope by moving its root up by one level (unless the root of the overal system has already been reached). Again, it needs to be ensured that no spurious transitions would be created from this step. This is done in lines 112–132 of the algorithm, again with the help of function TRYSYNC.

## 5 CORRECTNESS, EXHAUSTIVENESS AND COMPLEXITY OF THE ALGORITHM

**Correctness:** It must be guaranteed that a solution found by the algorithm is correct, which means that the modified compositional SPA system (with the calculated rates, possibly modified synchronisation sets and added selfloops) possesses exactly the same flat transition system as the original system, just with the transition rates modified as desired. In other words, correctness means that the modified flat transition system is structurally isomorphic to the original flat transition system, only the rates are changed to the desired values. Isomorphism of the flat transition systems implies that the two systems are functionally strongly bisimilar, but of course not strongly Markov bisimilar [10], since the rates have changed. Our proof of the algorithm's correctness is based on the following lemma:

LEMMA 5.1. *Let $Sys = P \mid\mid_A Q$ where $P, Q$ are sequential processes and $c \notin A$.*

(a) *(i) Assume that in the transition system of $P$ there is one single $c$-transition, namely $p \xrightarrow{c,\gamma_P} p'$, and likewise that in the transition system of $Q$ there is one single $c$-transition, namely $q \xrightarrow{c,\gamma_Q} q'$. (ii) Assume furthermore that in the low-level transition system of $Sys$ the combined states $\{(p, q_l) \mid l = 1, \ldots, L\}$ are reachable and that these are the only reachable states where $P$ is in state $p$ (other combined states where $P$ is not in state $p$ may also be reachable). Likewise, assume that the combined states $\{(p_m, q) \mid m = 1, \ldots, M\}$ are reachable and that these are the only reachable states where $Q$ is in state $q$. (iii) Assume also that combined state $(p, q)$ is not reachable.*

*Let $Q'$ be identical to $Q$ but with $L$ artificial $c$-selfloops added, namely the selfloops $q_l \xrightarrow{c,f_l} q_l$ where $l = 1, \ldots, L$. Likewise, let $P'$ be identical to $P$ but with $M$ artificial $c$-selfloops added, namely the selfloops $p_m \xrightarrow{c,g_m} p_m$ where $m = 1, \ldots, M$. Let $Sys' = P' \mid\mid_{A'} Q'$ where $A' = A \cup \{c\}$. Then the low-level transition systems of $Sys$ and $Sys'$ are structurally isomorphic.*

(b) *If, contrary to the assumptions in (a), there exists in $Q$ an outgoing non-selfloop $c$-transition from at least one of the states $q_1, \ldots, q_L$, for instance the transition $q_l \xrightarrow{c,\cdot} q'_l$, then the low-level transition systems of $Sys$ and $Sys'$ would not be structurally isomorphic. There is a symmetric argument in case there exists in $P$ an outgoing non-selfloop $c$-transition from at least one of the states $p_1, \ldots, p_M$.*

(c) *Let the situation be as in (a), with the exception that $P$ may now contain multiple $c$-transitions, namely $\tilde{p}_1 \xrightarrow{c,\gamma_{P,1}} \tilde{p}'_1$, $\tilde{p}_2 \xrightarrow{c,\gamma_{P,2}} \tilde{p}'_2, \ldots$, and that $Q$ may now contain multiple $c$-transitions, namely $\tilde{q}_1 \xrightarrow{c,\gamma_{Q,1}} \tilde{q}'_1$, $\tilde{q}_2 \xrightarrow{c,\gamma_{Q,2}} \tilde{q}'_2, \ldots$. Then, if all conditions named in (a) hold mutatis mutandis for each of those $c$-transitions, and if all the respective artificial selfloops in $Q$ and in $P$ were added, it follows that the low-level transition systems of $Sys$ and $Sys'$ are structurally isomorphic. If, however, similarly to part (b), in $Sys$ a combined state was reachble where both $P$ and $Q$ have an outgoing $c$-transition, then the low-level transition systems of $Sys$ and $Sys'$ would not be structurally isomorphic.*

Proof. Part (a): From the assumptions (i) ... (iii) it follows that $q$ is not equal to any of the $q_l$ and likewise that $p$ is not equal to any of the $p_m$. The original system $Sys$ has $L + M$ $c$-transitions, namely: $(p, q_l) \xrightarrow{c,\gamma_P} (p', q_l)$ where $l = 1, \ldots, L$, and $(p_m, q) \xrightarrow{c,\gamma_Q} (p_m, q')$ where $m = 1, \ldots, M$. The modified system $Sys'$ has exactly the same set of $c$-transitions. In $Sys'$ each of these transitions is realised by the $c$-transition in $P$ (in $Q$) which synchronises with the artificially inserted $c$-selfloop in one of the states $q_l$ (one of the $p_m$). All non-$c$-transitions in $Sys$ are not affected by the insertion of the artificial $c$-selfloops and the extra action $c$ in the synchronising set $A'$, therefore the non-$c$-transitions of $Sys$ and $Sys'$ are also identical. Thus, the low-level transition systems of $Sys$ and $Sys'$ are stucturally isomorphic. In case all of the values $f_l$ and $g_m$ are equal to 1, even the transition rates in $Sys$ and $Sys'$ are identical. Otherwise, by an appropriate choice of the values of $f_l$ and $g_m$ , different transition rates for the $c$-transitions in $Sys'$ can be realised.

Part (b): Notice that assumptions (i) ... (iii) in (a) imply that in $Q$ there is no outgoing $c$-transition from any one of the states $q_l$. Suppose now that $Q$ had a $c$-transition emanating from one of the states $q_l$, say transition $q_{l_1} \xrightarrow{c,\cdot} q'_{l_1}$. Since $(p, q_{l_1})$ is reachable, $Sys'$ would contain the transition $(p, q_{l_1}) \xrightarrow{c,\cdot} (p', q'_{l_1})$ which does not occur in $Sys$, thus being a spurious transition of type A. This proves part (b).

The proof of part (c) follows the same argumentation as that of parts (a) and (b), however requiring an elaborate notation because of the many possible combinations of states of $P$ and $Q$ that have to be considered. The simple case that $P$ (or $Q$) contains zero $c$-transitions is also included. □

Now we can argue constructively for the correctness of the algorithm as follows: In Part A, the Involved Set of transition $\hat{t}$ consists of only one single sequential process, denoted $P_{i_1}$. If the condition in line 21 of the algorithm is fulfilled, only the transition rate of a single transition in that processes needs to be changed, resulting in the simultaneous change of all the rates of the well-defined set of global flat "sibling" transitions $T_{c,\hat{s}_{i_1},\hat{s}'_{i_1}}$, all having the same modification factor, which yields the intended result. Obviously, this single rate change in Part A does not modify the structure of the underlying transition system. On the other hand, if the condition in line 21 is violated, the only chance of realising different rates for the transitions in the set $T_{c,\hat{s}_{i_1},\hat{s}'_{i_1}}$ is by augmenting the context and inserting controlling selfloops in other sequential processes, which is attempted in Part D. Parts B, C and D all rely on the fact that the rate of a flat transition is a function of the rates of transitions in all participating sequential processes. Consequently, to adjust the rate of $\hat{t}$ in the overall transition system, the rates in the participating processes $PS(\hat{t})$ must be set properly. Since these rates can have side-effects on other $c$-transitions in the same $c$-scope, all $c$-transitions in the scope $IS(\hat{t})$ must be considered at the same time in a system of equations. Therefore, Parts B, C and D each work by setting up and solving a nonlinear system of equations relating to the original (Part B) resp. carefully modified SPA system (Parts C and D). One equation is created for every transition $t$ in the set $T_c$, precisely reflecting the synchronisation of the sequential SPA processes in $PS(t)$ (see line 52 resp. line 90, which are both instances of Eq. 4). In the equation for transition $t$, since $t$ may be a collapsed multi-transition (cf. Sec. 3), all relevant selfloop combinations contributing to $t$ are reflected correctly by the summation over all elements of $rslc(t)$. The system of equations for $T_c$ takes into account all transitions with action label $c$ which belong to the current $c$-scope (which is equal to $IS(\hat{t})$), making sure that the resulting rates of those transitions are all as desired. If a solution for the unknown rates is found in Part B, the synchronisation sets remain untouched and no new selfloops are added, so obviously the structure of the underlying transition system remains the same. If in Parts C and D the model is adjusted (by augmenting one or more synchronisation sets and inserting artificial selfloops), Lemma 5.1 (applied analogously also to non-sequential processes) guarantees that this will not affect the structure of the low-level transition system, as far as spurious transitions of type A are concerned. In the pseudo-code, this is guaranteed by function TRYSYNC (see Appendix C), which also makes sure that no spurious transitions of type

B will be created by the adjustments taken, and also that all the transitions of the original flat transition system are still present. In other words, TRYSYNC guarantees that the original low-level transition system and the modified low-level transition system remain structurally isomorphic with each added $c$-synchronisation. In particular, during the checking for sp. tr. of type B, TRYSYNC computes feasible combinations $C^{feas}$ of processes synchronising on a $c$-transition, and is thus able to precisely determine the necessary new selfloops. Testing all nodes of type $X = X_1||_{\neg c}X_2$ in bottom-up order (line 76) is correct because it guarantees that sub-processes behave as expected, thus larger processes created from these building blocks will also behave as expected (this follows from the process algebra principle of compositionality [10]). In summary, since each individual step of the algorithm is correct, we can conclude by induction that the total effect of multiple steps is also correct.

**Exhaustiveness:** Related to the correctness of the algorithm, the question should be raised whether the algorithm uses all possible opportunities. In other words: If the lifting algorithm doesn't find a solution, is it really guaranteed that no solution exists? We now provide an informal proof of "exhaustiveness" in that sense: The algorithm's goal is to realise given rates in the low-level transition system, without stucturally changing the transition system. Since in a compositional SPA system the rate of a global transition $\hat{t}$ is a function of (the rate values of) all processes participating in that transition, the algorithm will have maximum freedom if it can set all contributing rates in the Participating Set $PS(\hat{t})$ freely. However, changing rates in processes from $PS(\hat{t})$ can have cross-effects on the rates of other transitions, as explained in Sec. 2.5. That's why all transitions with the same Involved Set $IS(\hat{t})$ need to be taken into account at the same time while processing $\hat{t}$, which is exactly what the algorithm does. If it turns out that the degrees of freedom inside $IS(\hat{t})$ are not sufficient for realising the desired rates, the algorithm augments the context of $\hat{t}$ (by synchronising with more processes) in the maximum permissible way. In detail, following the steps of the algorithm, we can give the basic line of argument for exhaustiveness: If we start with Part A of the algorithm and if local repair fails, this happens because the same local transition $\hat{s}_{i_1} \xrightarrow{c,\cdot} \hat{s}'_{i_1}$ (involving only a single sequential process $P_{i_1}$) should be executed in different contexts with different modification factors (i.e. different rates), which is of course not possible in the unmodified system. In order to solve this problem, some controlling context needs to be added. For this purpose, we synchronise the process with all possible neighbouring processes (where selfloops are added at specific states in order to partake in the existing transitions) in a subtree of a certain height, which leads to a set of equations (of the form of Eq. 4) in Part D of the algorithm. We keep expanding the context until either a solution has been found or the root of the system has been reached, which means that the algorithm has used the maximum potential. Alternatively, if we start with Part B (because the Involved Set of the currently processed transition $\hat{t}$ is already larger than one), we first search for a solution in the "local" context, i.e. in the current Involved Set, which is a subtree of the system. First we try to leave the model unchanged, which also leads to a system of equations. If it turns out that this system of equations is inconsistent (thus not having a solution), we need to include more degrees of freedom into the equations. This is first done within the current scope in Part C, by synchronising with as many processes as possible, albeit all from within this same scope. Note that there is no danger of inserting too many (permissible) selfloops, since the more combinations of selfloops exist, the more different rates are realisable. Therefore, our algorithm exhaustively includes all permissible such combinations of selfloops and thus does not miss any opportunity. If Part C also fails, i.e. if the thus extended system is also inconsistent, even more degrees of freedom can be added by expanding the current scope, leading us again to part D of the algorithm. In total, the algorithm uses all possible degrees of freedom, since at every step it involves all processes, except those whose involvement would cause damage (in the sense that spurious transitions would occur). So, in that sense, the algorithm is "exhaustive".

**Complexity:** We now analyse the asymptotic worst-case execution time complexity of the rate lifting algorithm. The analysis is not straightforward, since the algorithm's control structure opens up many possibe execution paths during which many subtasks have to be considered. The following notations are used: $n$ is the number of sequential processes in $Sys$. $S \subseteq S_1 \times \cdots \times S_n$ is the overall (combined) set of reachable states of $Sys$. $S_{max}$ is the set of states of the largest sequential process in $Sys$. $T$ is the set of (flat) transitions of $Sys$, where $|T| \leq |S|^2|Act|$. $T_{act=c} \subseteq T$ is the set of $c$-transitions. $T_{mod} \subseteq T$ is the set of transitions with modification factor $\neq 1$.

The process tree is an unbalanced binary tree. For $n$ sequential processes, the tree has $n$ leaf nodes and $n-1$ non-leaf nodes. Its depth is at most $n-1$. Traversal of the process tree takes $O(n)$ time. All neigbourhoods of processes (all $N_{type}(Sys, P, a)$, where $type \in \{cannot, must, may\}$) can be determined before running the lifting algorithm, but need to be updated every time the synchronisation structure is changed (when some node is made $c$-synchronising). Calculating the neighbourhoods of all processes (for a single action), each of which involves a (partial) process tree traversal, takes $O(nn) = O(n^2)$ time. All Moving Sets $MS(t)$ can be precalculated in time $O(|T|n)$ and will never change. All Participating Sets $PS(t)$ should also be precalculated, but they may change: If some node $X$ is made $c$-synchronising, then all $c$-transitions $t$ with $PS^{orig}(t) \cap X \neq \emptyset$ need to have $PS(t)$ updated (where $PS^{orig}(t)$ denotes the Participating Set of $t$ before the update). Calculating $PS(t)$ for a single transition $t$ (with the procedure from Sec. 2.4) takes $O(n^2)$ time. All Involved Sets $IS(t)$ can easily be precalculated, since they correspond to $c$-scopes in the process tree. During Part D of the algorithm, the Involved Sets may change, but they can be updated in $O(1)$ time.

If the algorithm succeeds in Part A, the complexity is dominated by the calculation of the set $T_{c,\hat{s}_{i_1},\hat{s}'_{i_1}}$, which takes at most $\alpha := |T_{act=c}| \leq |T|$ steps.

Running Part B of the algorithm is dominated by running the RSLC algorithm for each transition in $T_c$ and setting up and solving the system of nonlinear equations. The RSLC algorithm will create at most $2n$ recursive calls, of which the calls for the $n$ leaf nodes are the most expensive, since the set operations of line 7 (of RSLC) need to be executed, which takes $O(n)$. Since RSLC yields $O(2^n)$ combinations, the overal runtime of RSLC is $O(2nn + 2^n) = O(2^n)$. Therefore, the overall time for Part B is $O(\beta) := O(|T_c|2^n)$, plus the time for solving the nonlinear system of equations. We do not attempt to quantify the time for equation solving here, since this heavily depends on the algorithm and tool used.

The runtime for Part C is similar to that of Part B, but now the time for changing the synchronisation structure, done by function TRYSYNC, has to be added. TRYSYNC is called $O(\gamma_1) := O(n)$ times. During TRYSYNC, checking for spurious transitions of type A takes $O(\gamma_2) := O(|S|n)$ steps. Running function COMB, which is similar to RSLC (involving a process tree traversal, but now without the expensive set operations), produces $|C| \leq 2^n$ results, thus taking $O(\gamma_3) := O(2^n)$ steps. Therefore $C^{feas} \subseteq C$ also has at most $O(\gamma_4) := O(2^n)$ elements. For each combination $C_i \in C^{feas}$, there are $O(\gamma_5) := O(|T_c||S_{max}|n)$ new selfloops to be checked for and possibly inserted. For each new selfloop, checking for spurious transitions of type B takes $O(\gamma_6) := O(n|S|)$ steps (the process tree has to be traversed and transitions in $Y_2$ found). So the overall complexity for Part C is (considering the control structure, i.e. loops) $O(\gamma) := O(\gamma_1(\gamma_2 + \gamma_3 + \gamma_4\gamma_5\gamma_6) + \beta) = \cdots = O(n^3 2^n |T_c||S_{max}||S| + \beta) = O(n^3 2^n |T_c||S_{max}||S|)$ (remember that $\beta$ is the complexity of Part B).

The runtime complexity of Part D is the same as for Part C, since at the beginning function TRYSYNC is called once (for the new root of the $c$-scope), and then Parts B and possibly C are executed in that new context.

Summarising these partial results, the runtime of the rate lifting algorithm is the sum of the runtimes for Parts A to D, which is dominated by Parts C and D. Since the outermost while-loop of the algorithm (lines 13 - 129) performs at most $|T_{mod}|$ iterations, the overall runtime (excluding the time for solving the nonlinear system of equations) is $O(n^3 2^n |T_{mod}||T_c^{max}||S_{max}||S|)$, where $|T_c^{max}| = \max\{|T_c| : c \in Act\}$. It should be emphasised that this is the result of a pessimistic, worst-case analysis. In practice, the runtime of the core algorithm is moderate for small to medium-sized models (see Sec. 6).

As clearly stated above, our complexity analysis so far deliberately excluded the time for solving the nonlinear systems of equations. The time for equation solving may or may not dominate the overall runtime, depending on the particular problem instance at hand (see Sec. 6). If one wanted to include the time for equation solving in the complexity analysis, one would have to modify the expression for Part B to become $O(\beta) := O(|T_c|2^n + g(n, |T_c|, \dots))$, where $g$ is a yet unknown function of the number of components $n$, the number of $c$-transitions in the current $c$-scope $|T_c|$ and other (numerical) parameters. This modification would carry over to $O(\gamma)$ for Part C, which also depends on $\beta$. To the authors' knowledge, there are no generally valid results on the complexity of solving nonlinear systems of equations, although many studies of such systems exist, e.g. [9, 18, 23, 25]. The system of equations constructed by our algorithm (see eq. 4) has a special nonlinear form, called multilinear, because all unknowns appear as linear factors in product terms. For the special case $n = 2$, the paper [28] investigated efficient solutions of the associated bilinear system of equations. For the general case $n > 2$, however, a complexity analysis of the solving of the nonlinear system of equations is a difficult problem which must be left for future research.

## 6 EXPERIMENTAL RESULTS

### 6.1 Cyclic Server Polling System

This section considers – as a first case study – the Cyclic Server Polling System from the PRISM CTMC benchmarks, originally described in [12] as a GSPNs. It is a system where a single server polls $N$ stations and provides service for them in cyclic order. The SPA representation of this system is:

$$Sys = Server \;||_{\Sigma_s}\; (Station_1 \;||\; Station_2 \;||\; \dots \;||\; Station_N)$$

where $\Sigma_s = \{loop_{ia}, loop_{ib}, serve_i \mid i = 1 \dots N\}$ (actions with index $i$ synchronise between $Server$ and $Station_i$). A detailed description of the model is available in [12] and the SPA-style source code can be found at [22]. Assume that for each $loop_{1a}$-transition $t$ in the combined flat model a modification factor $f(t) \neq 1$ is given. From an application point of view, this assignment of modification factors aims at controlling the speed at which $Station_1$ is polled, depending on the global state of the system. Using our new lifting algorithm, we lift this model repair information to the components. The modification factors $f$ are chosen in such a way that local repair (Part A) and also Part B of the algorithm will not find a solution. In part C of the algorithm, it turns out that action $loop_{1a}$ can be added to all $||_{\neg loop_{1a}}$-nodes of the process tree, since it does not cause spurious transitions. Consequently, $loop_{1a}$-selfloops are added to all the states of the components $Station_2, \dots, Station_N$. This means that stations 2 to N, which were orignally not involved in action $loop_{1a}$, now also participate in that action. While this is not necessarily meaningful from an application point of view, it enables control of the rate of that action in a context-dependent fashion, as desired. These changes lead to the modified SPA model

$$Sys' = Server||_{\Sigma_s}(Station_1||_{loop_{1a}}Station'_2||_{loop_{1a}} \dots ||_{loop_{1a}}Station'_N)$$

where the stations with added selfloops are shown by $Station'_i$. With the chosen modification factors, a solution can be found in Part C of the algorithm. This example is a scalable model where the state space increases with the number of stations $N$. Note that the model contains cyclic symmetries, but it isn't lumpable in the sense of Markovian bisimulation (since the action names in different stations are differrent). The considered rate lifting problem, however, is not symmetric, since in the original model only $Station_1$ and the $Server$ participate in the $loop_{1a}$-transitions.

Table 1. Model statistics of the combined polling model for different numbers of stations

| N | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|
| Total number of states $\|S\|$ | 576 | 1344 | 3072 | 6912 | 15360 | 33792 |
| Total number of transitions $\|T\|$ | 2208 | 5824 | 14848 | 36864 | 89600 | 214016 |
| Number of $loop_{1a}$-transitions | 32 | 64 | 128 | 256 | 512 | 1024 |

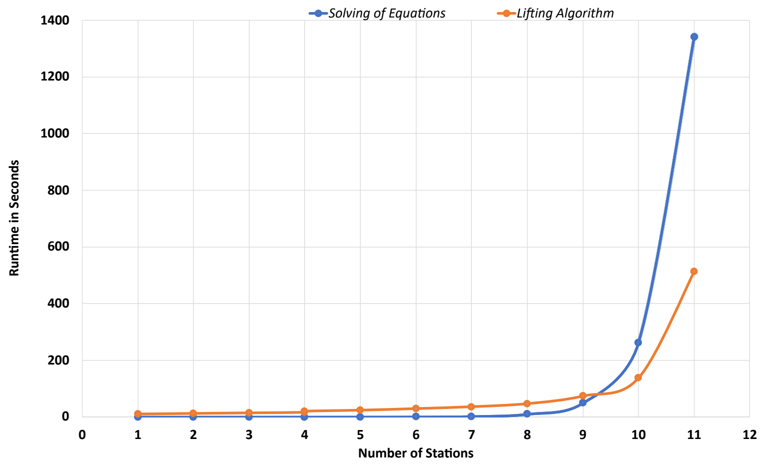

Fig. 5. Runtime comparison for different number of stations

Table 1 shows the model statistics for different numbers of stations. The last row of the table (number of $loop_{1a}$-transitions) equals the number of equations, each of the $2^{N-1}$ equations containing the product of $N+1$ unknown variables. The whole system of equations has $(N-1)*2+2$ variables stemming from $(N-1)*2$ newly added $loop_{1a}$-selfloops plus two original $loop_{1a}$-transitions (in the *Server* and $Station_1$). Figure 5 shows the required times to run our rate lifting algorithm (implemented as a proof-of-concept prototype in Matlab [14]) and to solve the system of equations (done by Wolfram Mathematica [15]) for different values of $N$ [3]. For this case study, as $N$ grows, the time for equation solving by far dominates the runtime of the lifting algorithm (by a factor of 2.61 for $N = 11$). As shown in the figure, the runtimes grow exponentially, which is not surprising since the number of equations increases exponentially.

## 6.2 Mobility-Aware RSVP

For this second case study, we consider a model of the Resource Reservation Protocol (RSVP) originally presented in [30]. In the modelled scenario, mobile nodes initiate communication sessions for which network resources need to be reserved. The network consists of the lower network channel (LNC) representing the mobile access network and the upper network channel (UNC) representing the Internet core network. LNC and UNC are queue-like processes with capacities $M$ resp. $N$, expressing resource limitations, where $M \geq N$. Blocking occurs when at the time of session initiation the requested resources are not available. When a session is initiated, resources both in LNC and UNC are requested, but when a handover occurs during a session, only resources in the LNC need to be requested since the

[3]Executed on a standard laptop with Intel Core i7-8650U CPU@ 1.90GHz-2.11GHz

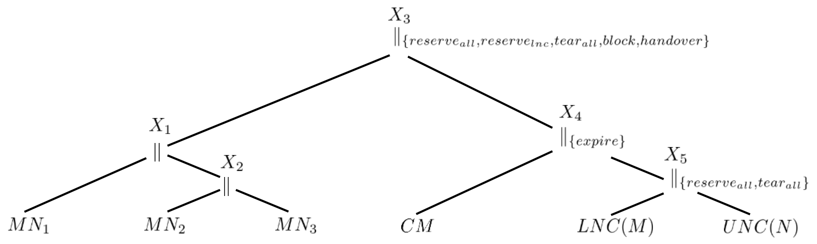
$X_3$
$\|_{\{reserve_{all}, reserve_{lnc}, tear_{all}, block, handover\}}$
$X_1$
$\|$
$X_2$
$\|$
$X_4$
$\|_{\{expire\}}$
$X_5$
$\|_{\{reserve_{all}, tear_{all}\}}$
$MN_1$
$MN_2$
$MN_3$
$CM$
$LNC(M)$
$UNC(N)$

Fig. 6. Process tree of the RSVP model (with three mobile nodes (MN))

Table 2. Model statistics of different configurations of the RSVP model

| | | | lifting for action *handover* | | | lifting for action *expire* | | |
|---|---|---|---|---|---|---|---|---|
| model config. | #states | #trans | #eqns | algo (sec) | solve (sec) | #eqns | algo (sec) | solve (sec) |
| MN3M5N3 | 592 | 2576 | 78 | 26.22 | 0.44 | 467 | 52.68 | 3.13 |
| MN3M7N5 | 842 | 3661 | 116 | 32.30 | 0.58 | 717 | 67.52 | 4.13 |
| MN4M7N5 | 3984 | 21756 | 240 | 242.20 | 1.08 | 3359 | 1307.17 | 72.91 |
| MN5M5N5 | 12468 | 79468 | 275 | 1872.37 | 1.19 | 9343 | 7344.45 | 274.76 |
| MN5M7N5 | 18718 | 123566 | 425 | 4520.75 | 5.04 | 15593 | 40358.23 | 1298.25 |

already allocated resources in the UNC can be kept. The model contains one process for each mobile node (MN), for the lower and upper network channel (LNC and UNC) and for the so-called channel monitor (CM). The latter is an auxiliary process needed to ensure that resources no longer needed in LNC are properly released. Fig. 6 shows the process tree of the SPA model for an instance with three MNs, where the synchronisation sets are also indicated.

To test our rate lifting algorithm, we considered five different configurations of the RSVP model. Configuration MN$x$M$y$N$z$ denotes a model with $x$ mobile nodes and queue sizes $y$ resp. $z$. The model statistics for the different configurations are shown in Table 2, where the second / third column gives the number of overall states / flat transitions of the model.

We start by setting rate modification factors for all *handover*-transitions of the flat transition system. In the original model, the Participating Set of each *handover* transition consists of exactly one MN and the CM process. That means that the algorithm starts with Part B. However, we set the modification factors in such a way that the system of equations in Part B has no solution. Thus the algorithm proceeds with Part C, where it turns out that *handover* can be added to the synchronisation set of $X_4$, resulting in artificial selfloops in processes UNC and LNC, leading to a system of equations which does possess a solution. The relevant statistics are given in columns 4 to 6 of Table 2: Column 4 gives the number of unique equations (since a *handover* transition with given Participating Set may take place in different contexts, sometimes possibly with the same modification factor, some of the constructed equations may be identical. In general, for $a \in Act$, if distinct $a$-transitions have the same Participating Set and the same modification factor, this results in identical equations). Column 5 of the table is the execution time of the lifting algorithm, and column 6 is the Matlab solution time for the system of equations[4]. Interestingly, here the overall time is dominated by the execution time of the lifting algorithm and not by the numerical equation solving. The reason is that – contrary to the polling example in Sec. 6.1 – we chose "benignant" modification factors which admit a solution without running into numerical difficulties.

[4]Executed again on the same standard laptop with Intel Core i7-8650U CPU@ 1.90GHz-2.11GHz

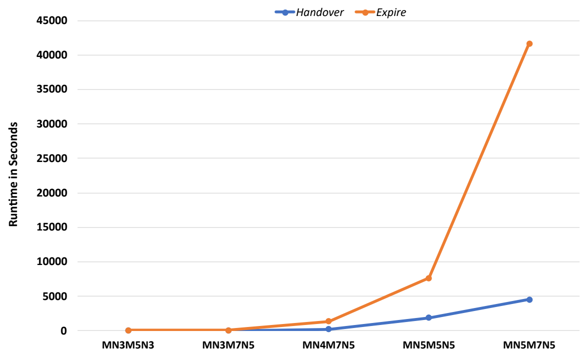

Fig. 7. Runtime for different configurations of the RSVP model

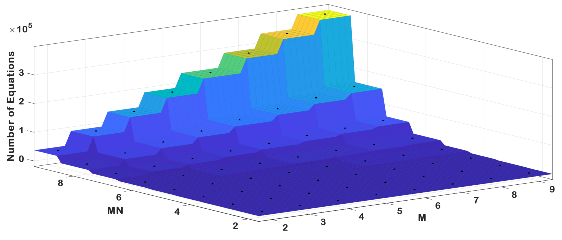

Fig. 8. Number of equations for action *expire* in the RSVP model

We proceed by setting modification factors for all *expire*-transitions. Notice that the *expire*-scope of the RSVP model consists of the subtree rooted at $X_4$. Only processes LNC and CM participate in the *expire* action (UNC does not). The modification factors were chosen in such a way that Part B of the algorithm fails. In Part C, action *expire* can be successfully added to node $X_5$ of the process tree, thereby adding *expire* selfloops to UNC without causing spurious transitions, but unfortunately still the system of equations does not possess any solution. The algorithm then moves on to Part D, which succeeds by adding *expire* to the synchronisation set of $X_3$ (i.e. the needed selfloops in $\text{MN}_1$, $\text{MN}_2$ and $\text{MN}_3$ do not cause spurious transitions) and finding a solution for the resulting system of equations. The relevant statistics are given in columns 7 to 9 of Table 2. Now the runtime of the lifting algorithm becomes really exorbitant, on our modest hardware it is more than 11 hours for the largest model!

Fig. 7 visualises the runtimes of the algorithm (including numerical solution) for different RSVP model instances for both the *handover* (blue) and *expire* (orange) actions. Finally, Fig. 8 shows the number of equations constructed by the lifting algorithm (in Part D) when handling the *expire*-action, for values of $MN$ and $M$ beyond those considered in Table 2 (note that $N$ has no influence on the number of equations for action *expire*). For MN = 9 and M = 9 we would need to solve a nonlinear system with more than 350000 equations! This is, of course, beyond the scope of what can be done on standard hardware.

## 7 CONCLUSION

In this paper, we have presented a novel algorithm for the lifting of rate information from the flat low-level transition system of an SPA model to its components. The algorithm works for general SPA models with an arbitrary structure and any number of components. We have established some novel structural and behavioural concepts of Markovian SPA, which were needed to formulate the lifting algorithm. A key idea of the algorithm is the augmentation of a given flat transition's context, by synchronising with additional components and inserting artificial selfloops, thereby enhancing the control over a transition's rate. While doing this, the algorithm takes great care to avoid negative side effects such as spurious transitions. The algorithm ensures that the modifications of the SPA model do not change its qualitative behaviour – only the transition rates are modified as desired. During the algorithm, systems of nonlinear equations for the unknown component rates are constructed. For their solution, instead of developing our own methods, for the moment we make use of available numerical solvers, since the details of nonlinear equation solving are beyond the scope of the present paper. It would, however, be an interesting research problem to develop efficient numerical solution schemes dedicated to this particular type of nonlinear equations. It is also important to mention that the solution of this type of nonlinear system of equations, in case such a solution exists, is in general not unique. In case several different solutions exist, the question arises which of them is optimal, but this is a non-trivial question since optimality criteria are yet to be defined. Therefore, the search for an optimal solution is also an interesting topic for future research. Apart from a thorough analysis of the algorithm regarding its correctness and complexity, the paper also presented two case studies that illustrate the successful practical use of the algorithm. As future work, we are also planning to develop improved implementation strategies for the algorithm, based on compact data structures. Another important point for future work is to characterise a priori the set of problem instances for which a solution to the rate lifting problem exists.

## REFERENCES

[1] C. Baier, B. R. Haverkort, H. Hermanns, and J.-P. Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE Trans. Software Eng.*, 29(6):524–541, 2003.

[2] E. Bartocci, R. Grosu, P. Katsaros, C. R. Ramakrishnan, and S.A. Smolka. Model repair for probabilistic systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 326–340. Springer, LNCS 6605, 2011.

[3] M. Bernardo. *Theory and Application of Extended Markovian Process Algebra.* PhD thesis, University of Bologna, 1999.

[4] C. Cai, J. Sun, G. Dobbie, Z. Hóu, H. Bride, J. Dong, and S. Lee. Fast automated abstract machine repair using simultaneous modifications and refactoring. *Form. Asp. Comput.*, 34(2):8:1–8:31, 2022.

[5] T. Chen, E.M. Hahn, T. Han, M. Kwiatkowska, H. Qu, and L. Zhang. Model repair for Markov decision processes. In *2013 International Symposium on Theoretical Aspects of Software Engineering*, pages 85–92, 2013.

[6] C. Dehnert, S. Junges, J.-P. Katoen, and M. Volk. A storm is coming: A modern probabilistic model checker. In R. Majumdar and V. Kunčak, editors, *Computer Aided Verification*, pages 592–600, Cham, 2017. Springer, LNCS 10427.

[7] N. Götz. *Stochastische Prozessalgebren: Integration von funktionalem Entwurf und Leistungsbewertung Verteilter Systeme.* PhD thesis, University of Erlangen-Nuremberg, Germany, 1994.

[8] A. Gouberman, M. Siegle, and B.S.K. Tati. Markov chains with perturbed rates to absorption: Theory and application to model repair. *Performance Evaluation*, 130:32–50, 2019.

[9] M. Grau-Sanchez, A. Grau, and M. Noguera. On the computational efficiency index and some iterative methods for solving systems of nonlinear equations. *Journal of Computational and Applied Mathematics*, 236(6):1259–1266, 2011.

[10] H. Hermanns, U. Herzog, and J.-P. Katoen. Process algebra for performance evaluation. *Theoretical Computer Science*, 274(1):43–87, 2002.

[11] J. Hillston. *A Compositional Approach to Performance Modelling.* Cambridge University Press, New York, NY, USA, 1996.

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

[13] Gurobi Optimization, Inc. Gurobi, Version 9.5, 2022.

[14] The MathWorks, Inc. Matlab, Version R2022a, 2022.

[15] Wolfram Research, Inc. Mathematica, Version 12.2.0.0. Champaign, IL, 2021.

[16] M. Kuntz, M. Siegle, and E. Werner. Symbolic Performance and Dependability Evaluation with the Tool CASPA. In *Applying Formal Methods: Testing, Performance and M/E Commerce: FORTE 2004 Workshops, European Performance Engineering Workshop*, pages 293–307. Springer, LNCS 3236, 2004.

[17] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[18] J. M. Martínez. Algorithms for solving nonlinear systems of equations. In Emilio Spedicato, editor, *Algorithms for Continuous Optimization: The State of the Art*, pages 81–108. Springer Netherlands, 1994.

[19] J. Meyer. On evaluating the performability of degradable computing systems. *IEEE Transactions on Computers*, C-29(8):720–731, 1980.

[20] R. Milner. *A Calculus of Communicating Systems.* Springer, LNCS 92, 1980.

[21] S. Pathak, E. Ábrahám, N. Jansen, A. Tacchella, and J.-P. Katoen. A greedy approach for the efficient repair of stochastic models. In K. Havelund, G. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, pages 295–309, Cham, 2015. Springer, LNCS 9058.

[22] PRISM. Cyclic server polling system CTMC case study at https://www.prismmodelchecker.org/casestudies/polling.php. Accessed on Sept. 14, 2023.

[23] J. R. Sharma, R. K. Guha, and R. Sharma. An efficient fourth order weighted-Newton method for systems of nonlinear equations. *Numerical Algorithms*, 62:307–323, 2013.

[24] M. Siegle and A. Soltanieh. Rate lifting for stochastic process algebra – exploiting structural properties –. In E. Ábrahám and M. Paolieri, editors, *Quantitative Evaluation of Systems*, pages 67–84, Cham, 2022. Springer, LNCS 13479.

[25] K. Sikorski. Optimal solution of nonlinear equations. *Journal of Complexity*, 1(2):197–209, 1985.

[26] A. Soltanieh. *Compositional Stochastic Process Algebra Models: A Focus on Model Repair and Rate Lifting.* PhD thesis, Universität der Bundeswehr München, Dept. of Computer Science, 2022.

[27] A. Soltanieh and M. Siegle. It sometimes works: A lifting algorithm for repair of stochastic process algebra models. In *Measurement, Modelling and Evaluation of Computing Systems*, pages 190–207. Springer, LNCS 12040, 2020.

[28] A. Soltanieh and M. Siegle. Solving systems of bilinear equations for transition rate reconstruction. In H. Hojjat and M. Massink, editors, *Fundamentals of Software Engineering*, pages 157–172, Cham, 2021. Springer, LNCS 12818.

[29] B.S.K. Tati and M. Siegle. Rate reduction for state-labelled Markov chains with upper time-bounded CSL requirements. In T. Brihaye, B. Delahaye, L. Jezequel, N. Markey, and J. Srba, editors, Proceedings Cassting Workshop on *Games for the Synthesis of Complex Systems* and 3rd International Workshop on *Synthesis of Complex Parameters,* Eindhoven, The Netherlands, April 2-3, 2016, volume 220 of *Electronic Proceedings in Theoretical Computer Science*, pages 77–89. Open Publishing Association, 2016.

[30] H. Wang, D.I. Laurenson, and J. Hillston. Evaluation of RSVP and mobility-aware RSVP using performance evaluation process algebra. In *IEEE International Conference on Communications (ICC)*, pages 192–197. IEEE, 2008.

## A RATE LIFTING ALGORITHM FOR AN SPA SYSTEM WITH $n$ SEQUENTIAL COMPONENTS

```
1: **Algorithm** RateLifting ($Sys, T, T_{mod}, factor$)
2: // $T$ is the flat Markovian transition system of SPA system $Sys$,
3: // consisting of sequential processes $P_1, \dots, P_n$ as leaves of a process tree with synchronisation sets $A_i$.
4: // The algorithm lifts the repair information given in the form of rate modification factors $factor(t)$ for
5: // transitions $t \in T_{mod} \subseteq T$ to the high-level components of $Sys$, if possible.
6: // The repaired system is returned as $P'_1, \dots, P'_n$ and possibly modified synchronisation sets $A'_i$.
7:
8: $P'_1 := P_1, \dots, P'_n := P_n$, $\forall i : A'_i := A_i$ // initialisation
9:
10: **while** $T_{mod} \neq \emptyset$ **do**
11:     choose $\hat{t} := ((\hat{s}_1, \dots, \hat{s}_n) \xrightarrow{c,\hat{\gamma}\cdot\hat{f}} (\hat{s}'_1, \dots, \hat{s}'_n))$ from $T_{mod}$
12:     // $\hat{t}$ is the transition processed during one iteration of the outer while-loop
13:     $found := false$
14:     // indicates that no solution was found yet while processing the current $\hat{t}$
15:     determine $IS(\hat{t}) := \{P_{i_1}, \dots, P_{i_m}\}$ // the Involved Set $IS$ is also a $c$-scope
16:
17:     **if** $|IS(\hat{t})| = 1$ **then**
18:         // **Algorithm Part A:**
19:         // try local repair in $P_{i_1}$ by considering all "sibling" transitions
20:         $T_{c,\hat{s}_{i_1},\hat{s}'_{i_1}} := \{t \in T \mid action(t) = c \wedge source_{i_1}(t) = \hat{s}_{i_1} \wedge target_{i_1}(t) = \hat{s}'_{i_1}\}$
21:         **if** $\exists f_{com} \in \mathbb{R} : \forall t \in T_{c,\hat{s}_{i_1},\hat{s}'_{i_1}} : factor(t) = f_{com}$ **then**
22:             // there exists a common factor $f_{com}$ for all transitions in $T_{c,\hat{s}_{i_1},\hat{s}'_{i_1}}$
23:             in $P'_{i_1}$ set $\hat{s}_{i_1} \xrightarrow{c,\gamma_{i_1}\cdot f_{com}} \hat{s}'_{i_1}$ (where $\gamma_{i_1}$ is the current rate in $P'_{i_1}$)
24:             $T_{mod} := T_{mod} \setminus T_{c,\hat{s}_{i_1},\hat{s}'_{i_1}}$
```

```
25:         for each $t \in T_{c,\hat{s}_{i_1},\hat{s}'_{i_1}}$ do
26:             $factor(t) := 1$
27:             // the modification factor of the fixed transitions is changed to 1,
28:             // which is important in case they are considered again
29:             // when dealing with another $c$-transition from $T_{mod}$ later
30:         end for
31:         $found = true$
32:     end if
33:     if $found = false$ then
34:         // local repair was not successful
35:         // therefore, since $|IS(\hat{t})| = 1$, the algorithm has to move up in the tree
36:         $curr\_root := P_{i_1}$
37:         // $curr\_root$ denotes the root of the subtree that is currently
38:         // considered as the context for transition $\hat{t}$
39:     end if
40:
41: else
42:     // Algorithm Part B:
43:     // it holds that $|IS(\hat{t})| > 1$
44:     $T_c := \{t \in T \mid action(t) = c \ \wedge \ IS(t) = IS(\hat{t})\}$
45:     // all $c$-transitions in current $c$-scope are considered together
46:     $r$ := root of current $c$-scope // needed in lines 49 and 84
47:     for each $t := ((s_1, \ldots, s_n) \xrightarrow{c,\gamma \cdot f} (s'_1, \ldots, s'_n)) \in T_c$ do
48:         determine $PS(t) := \{P_{p_1}, \ldots, P_{p_k}\} \subseteq IS(\hat{t})$
49:         find the set of relevant selfloop combinations $\mathrm{rslc}(t) := \mathrm{RSLC}(t, r)$
50:         // for rslc see Sec. 3.2 and Appendix B
51:         create an equation
52:         $\sum_{C \in rslc(t)} \prod_{P \in MS(t)} x^{(P)}_{s_P s'_P} \prod_{\substack{Q \in PS(t) \setminus MS(t) \\ \wedge\ \exists P \in MS(t):\ Q \in N_{must}(Sys,P,c)}} x^{(Q)}_{s_Q s_Q} \prod_{R \in C} x^{(R)}_{s_R s_R} = \gamma \cdot f$
53:     end for
54:     solve system of equations, if successful set $found := true$
55:     if $found = true$ then
56:         use the found solution to set the rates in the sequential processes accordingly
57:         $T_{mod} := T_{mod} \setminus T_c$
58:         for each $T \in T_c$ do
59:             $factor(t) := 1$
60:             // needed in case same transition is considered again later
61:         end for
62:
63:     else
64:         // Algorithm Part C:
65:         // it still holds that $found = false$
66:         // now try to change the model:
67:         if $PS(\hat{t}) \neq IS(\hat{t})$ then
```

```
68:              // the condition $PS(\hat{t}) \neq IS(\hat{t})$ means that there exist processes in the current $c$-scope which
69:              // could possibly be added to $PS(\hat{t})$
70:              // the algorithm tries to add $c$ to the sync. set of every internal node $X$
71:              // of the form $X = X_1||_{\neg c}X_2$, within the current $c$-scope,
72:              // but this must not lead to sp. tr. outgoing from a reachable state.
73:              // If $c$ can be added, then the necessary $c$-selfloops must also be added.
74:              $success := false$
75:              // variable $success$ remembers if at least one of the following calls to TRYSYNC (line 77) returns true
76:              for all nodes $X = X_1||_{\neg c}X_2$ in current $c$-scope (in bottom-up order) do
77:                 $success := success \vee$ TRYSYNC $(X, c)$
78:                 // TRYSYNC tries to add $c$ to the sync. set of node $X$,
79:                 // if possible it changes $X$ to $||_c$, adds necessary selfloops below $X$ and returns true
80:              end for
81:              if $success$ then
82:                 for each $t := ((s_1, \ldots, s_n) \xrightarrow{c,\gamma\cdot f} (s'_1, \ldots, s'_n)) \in T_c$ do
83:                    determine $PS(t) := \{P_{p_1}, \ldots, P_{p_k}\} \subseteq IS(\hat{t})$
84:                    find the set of relevant selfloop combinations rslc$(t) := RSLC(t, r)$
85:                    // similar calculations as in the two previous lines
86:                    // have already been performed in lines 48 and 49, but now the
87:                    // model is changed, so $PS(t)$ will potentially be larger than before,
88:                    // and the sets in rslc$(t)$ will potentially be larger than before!
89:                    create an equation
90:                    $\sum_{C\in rslc(t)} \prod_{P\in MS(t)} x^{(P)}_{s_P s'_P} \prod_{\substack{Q\in PS(t)\setminus MS(t) \\ \wedge\ \exists P\in MS(t):\ Q\in N_{must}(Sys,P,c)}} x^{(Q)}_{s_Q s_Q} \prod_{R\in C} x^{(R)}_{s_R s_R} = \gamma\cdot f$
91:                 end for
92:                 solve system of equations, if successful set $found := true$
93:                 if $found = true$ then
94:                    // changing the model led to a solution
95:                    use the found solution to set the rates in the sequential processes accordingly
96:                    $T_{mod} := T_{mod} \setminus T_c$
97:                    for each $T \in T_c$ do
98:                       $factor(t) := 1$
99:                       // needed in case same transition is considered again later
100:                   end for
101:                else
102:                   // no solution found in current $c$-scope, even with added selfloops
103:                   // it still holds that $found = false$
104:                   $curr\_root :=$ root of current $c$-scope
105:                end if
106:             end if
107:          end if
108:       end if
109:    end if
110:
111:   // Algorithm Part D:
```

```
112:   while $found = false$ and $curr\_root \neq root(Sys)$ do
113:     // move current root upwards and try to expand $c$-scope
114:     $curr\_root := parent(curr\_root)$
115:     $success$ := TRYSYNC($curr\_root$, $c$)
116:     // TRYSYNC tries to add $c$ to the sync. set of node $curr\_root$,
117:     // and if possible changes $curr\_root$ to $||_c$ and adds the
118:     // necessary selfloops below $curr\_root$
119:     if $success$ then
120:       $IS_{new}(\hat{t})$ := all leaves of subtree rooted at $curr\_root = \{P_{i_1}, \ldots, P_{i_m}\}$
121:       // this $m$ is now larger than in line 15,
122:       // it gets larger in each iteration of this while-loop
123:       $IS(\hat{t}) := IS_{new}(\hat{t})$
124:       $T_c := \{t \in T \mid action(t) = c \;\wedge\; PS(t) \cap IS(\hat{t}) \neq \emptyset\}$
125:       $r := curr\_root$ // root of current $c$-scope
126:       now the same code as in lines 47–108
127:       ...
128:     else
129:       BREAK while-loop of lines 112–132 and move $curr\_root$ up further
130:       // it was not possible to add $c$ to the sync. set at $curr\_root$
131:     end if
132:   end while
133: end while
```

## B RSLC ALGORITHM (RELEVANT SELFLOOP COMBINATIONS)

Note: RSLC is called from the main rate lifting algorithm in lines 49 and 84.

```
1:  Algorithm RSLC ($t, n$)
2:  // $t \in T$ is a transition
3:  // $n$ is a node of the process tree of $Sys$, denoting the current position in the recursive descent
4:  // The algorithm returns a set of sets of sequential processes
5:  // (each representing a relevant selfloop combination contributing to $t$)
6:  if type($n$) = leaf then
7:    if $P_n \in (PS(t) \cap SS(t)) \setminus \bigcup_{P \in MS(t)} N_{must}(Sys, P, c)$ then
8:      // $P_n$ denotes the process represented by leaf-node $n$
9:      // $P_n \in PS(t)$ ensures that $P_n$ has a selfloop at its current state $source_{P_n}(t)$
10:     return $\{\{P_n\}\}$
11:     // a set containing a singleton set is returned
12:   else
13:     return $\{\emptyset\}$
14:     // the set containing the empty set is returned
15:   end if
16: else if type($n$) = $||_c$ then
```

```
17:     return $\{C_1 \cup C_2 \mid C_1 \in RSLC(t, \text{lchild}(n)) \wedge C_2 \in RSLC(t, \text{rchild}(n))\}$
18:     // all combinations of left and right subtree
19: else
20:     // it holds that type($n$) = $||_{\neg c}$
21:     return $\{C \mid C \in RSLC(t, \text{lchild}(n)) \vee C \in RSLC(t, \text{rchild}(n))\}$
22:     // the (disjoint) union of left and right subtree
23: end if
```

## C TRYSYNC FUNCTION

Note: TRYSYNC is called from the main algorithm in lines 77 and 115.

```
1: Function: TRYSYNC $(X, c)$ returns Boolean
2: // This function checks whether adding action $c$ to node $X$ of type $||_{\neg c}$
3: // is possible without creating spurious transitions.
4: // If possible, it changes $X$ to $||_c$ and adds the necessary selfloops.
5: // The return value is true iff action $c$ could be successfully added.
6: if changing $X$ to $||_c$ would cause sp. tr. of type A then
7:     // checking for sp. tr. of type A can be done
8:     // by ensuring that for all reachable states $\overrightarrow{s} = (s_1, ..., s_n)$,
9:     // transitions $\overrightarrow{s}_{X_1} \xrightarrow{c} \ldots$ and $\overrightarrow{s}_{X_2} \xrightarrow{c} \ldots$ do not both exist,
10:    // where $X_1$ and $X_2$ are the children of node $X$.
11:    return false // do nothing, since $X$ can't be made $c$-synchronising
12: end if
13: compute $C := COMB(X, c)$
14: // $C$ contains all possible combinations of $c$-participants below $X$, assuming $X$ was of type $||_c$
15: $T_c^X = \{t \in T_c \mid PS(t) \cap P_X \neq \emptyset\}$
16: // where $P_X$ denotes the set of all sequential processes contained in $X$,
17: // so $T_c^X$ is the subset of $c$-transitions with some process from $X$ participating
18: // compute the set of feasible combinations $C^{feas} \subseteq C$:
19: $C^{feas} := \{C_i \in C \mid \forall t \in T_c^X$ :
20:             (selfloops in all $P_k \in C_i \setminus PS(t)$ at state $source_k(t)$ are present
21:             or can be added without causing sp. tr. of type B)$\}$
22: if $C^{feas} = \emptyset$ then
23:     return false // do nothing, since $X$ can't be made $c$-synchronising (because there doesn't exist
24:     // any feasible combination)
25: end if
26: // now we know that $X$ can be made $c$-synchronising!
27: change $X$ to type $||_c$ // add action $c$ to the sync. set $A_i$ corresponding to $X$
28: // now permanently add the new selfloops:
29: for each $C_i \in C^{feas}$ do
30:     for each $t \in T_c^X$ do
```

```
31:         // it suffices to look at projection $t^X : \overrightarrow{s_X} \xrightarrow{c} \overrightarrow{s_X}'$
32:         for each $P_k \in C_i \wedge P_k \notin PS(t)$ do
33:             if selfloop in $P_k : s_k \xrightarrow{c} s_k$ does not exist then
34:                 add selfloop in $P_k : s_k \xrightarrow{c, x^{P_k}_{s_k, s_k}} s_k$
35:             end if
36:         end for
37:     end for
38: end for
39: return  true // $X$ has been changed to $||_c$ and the new selfloops added
40: end function
```

**Checking for Spurious Transitions of Type B:** We now discuss the details of how it can be checked whether a combination $C_i \in C = COMB(X, c)$ is feasible or not, i.e. the details of lines 19 - 21 of function TRYSYNC. We are in the process of checking whether making node $X = X_1 ||_{\neg c} X_2$ $c$-synchronising (and adding some selfloops as needed by combination $C_i$) causes sp. tr. of type B or not. The following program segment performs this check:

```
1:  assume (temporarily) that $X = X_1 ||_c X_2$     // i.e. assume that $X$ was $c$-synchronising
2:  // compute the set of newly needed selfloops for combination $C_i$:
3:  $Selfloops(C_i) := \{(P_k, s_{k_j}) \mid \exists t \in T^X_c : P_k \in C_i \setminus PS(t) \wedge s_{k_j} = source_{P_k}(t)$
4:                                  $\wedge$ selfloop $s_{k_j} \xrightarrow{c} s_{k_j}$ does not yet exist$\}$
5:  for all $P_k$ where any new selfloops are needed for $C_i$ do
6:      // assume that $P_k$ is part of $X_1$, otherwise symmetric procedure
7:      let $Y := Y_1 ||_c Y_2$ be the lowest $c$-synchronising node containing $P_k$
8:      // assume that $P_k$ is part of $Y_1$, otherwise symmetric procedure
9:      // initially, $Y$ is either part of $X_1$, or $Y = X$ (since $X$ is now $c$-synchronising)
10:     // later (when moving $Y$ upwards, see line 29), $Y$ can be even above $X$
11:     for all states $s_{k_j}$ where a selfloop is needed and not yet present in $P_k$ do
12:         temporarily add selfloop $s_{k_j} \xrightarrow{c} s_{k_j}$
13:         set $Z := P_k$ and $\overrightarrow{z} := s_{k_j}$
14:         // we use $Z$ to denote a subsystem with a selfloop and $\overrightarrow{z}$ its state
15:         // initially, $Z$ is equal to only $P_k$, but when moving $Y$ upwards (see line 29)
16:         // $Z$ will be a larger subsystem
17:         if $Y \neq X$ and $\exists \overrightarrow{s}$ reachable state such that $\overrightarrow{s}_Z = \overrightarrow{z}$ and $\exists \overrightarrow{s}_{Y_2} \xrightarrow{c} \ldots$ then
18:             // non-selfloop $c$-transition in $\overrightarrow{s}_{Y_2}$ would synchronise with new (atomic or combined) selfloop at $\overrightarrow{z}$
19:             // which means that a sp. tr. of type B exists, therefore $C_i$ is not feasible
20:             remove all selfloops newly added while processing $X$
21:             BREAK // because $C_i$ is not feasible
22:         else if $\exists \overrightarrow{s}$ reachable state such that $\overrightarrow{s}_Z = \overrightarrow{z}$ and $\exists \overrightarrow{s}_{Y_2} \xrightarrow{c} \overrightarrow{s}_{Y_2}$ then
23:             // selfloop $c$-transition in $\overrightarrow{s}_{Y_2}$ would synchronise with new (atomic or combined) selfloop at $\overrightarrow{z}$,
24:             // yielding a new combined selfloop $\overrightarrow{s}_Y \xrightarrow{c} \overrightarrow{s}_Y$
25:             // having found such a new combined selfloop, it has to be ensured that it will not cause
```

```
26:         // a sp. tr. of type B by synchronising with another process higher up in the tree,
27:         // therefore $Y$ has to be moved upwards
28:         move upwards    // i.e. rerun the if-clause (lines 17 - 33) for $Z := Y, \vec{z} := \vec{s}_Y$ and
29:                         // $Y :=$ next higher node $Y$ of type $||_c$, if such a higher node $Y$ exists
30:       else
31:         // no $c$-transition nor $c$-selfloop in $\vec{s}_{Y_2}$
32:         // selfloop can be safely added to $s_{k_j}$ without causing sp. tr. of type B
33:       end if
34:   end for
35: end for
36: // once both FOR-loops have terminated without BREAK, we know that combination $C_i$ is feasible
```

## D COMB ALGORITHM

Note: COMB is called from TRYSYNC in line 13.

```
1: Algorithm COMB ($X, c$)
2: // This algorithm returns all sequential component combinations under
3: // node $X$ wrt action $c$. Every combination consists of some sequential
4: // components that may participate together in a $c$-transition.
5: // Like RSLC, this algorithm returns a set of sets of sequential processes.
6: if type(X)=leaf then
7:   return {{$P_X$}}
8:   // the seq. process
9: else if type($X$)=$||_c$ then
10:   return
11:   $\{C_1 \cup C_2 \mid C_1 \in \text{COMB}(\text{lchild}(X), c) \wedge C_2 \in \text{COMB}(\text{rchild}(X), c)\}$
12: else
13:   //type(X)=$||_{\neg c}$
14:   return
15:   $\{C \mid C \in \text{COMB}(\text{lchild}(X), c) \vee C \in \text{COMB}(\text{rchild}(X), c)\}$
16: end if
```