Lumping partially symmetrical stochastic models

S. Baarir[a], M. Beccuti[b], C. Dutheillet[a], G. Franceschinis[c], S. Haddad[d]

[a]*LIP6, Université Paris 6*
[b]*Dip. di Informatica, Univ. di Torino*
[c]*Dip. di Informatica, Univ. Piemonte Orientale*
[d]*LSV, ENS Cachan*

Abstract

Performance and dependability evaluation of complex systems by means of dynamic stochastic models (e.g. Markov chains) may be impaired by the combinatorial explosion of their state space. Among the possible methods to cope with this problem, symmetry-based ones can be applied to systems including several similar components. Often however these systems are only partially symmetric: their behaviour is in general symmetric except for some local situation when the similar components need to be differentiated.

In this paper two methods to efficiently analyze partially symmetrical models are presented in a general setting and the requirements for their efficient implementation are discussed. Some case studies are presented to show the methods effectiveness and their applicative interest.

Keywords: Markov chains, lumpability, stochastic Petri nets, symmetries.

1. Introduction

As software systems and hardware architectures are more and more complex, their verification and evaluation become critical issues. Analysis methods are often subject to the problem of combinatorial explosion due to the increasing system complexity. Several approaches have been undertaken to cope with this problem: decomposition methods take advantage of the modular structure of the system; for performance evaluation, approximate and bounding methods substitute a simpler system to the original one; diagram decision based methods symbolically manage sets of states rather than representing states explicitly, etc. Here, we present symmetry-based methods that exploit the presence of several similar components in the system.

The general principle of these methods consists to substitute to the state graph a quotient graph w.r.t. some equivalence relation. This relation considers two states as equivalent if they can be obtained from each other permuting equivalent components. These methods have been first introduced in order to check safeness properties (see e.g. [1, 2]), then generalized in order to check temporal logic formulae (see e.g. [3]) and also adapted to performance evaluation (through a Markov chain) *via* the quantitative counterpart of symmetry, i.e. *lumpability* (see e.g. [4, 5, 6, 7]). It should be stressed that the requirements w.r.t. lumpability are generally stronger than the ones that ensure equivalence between qualitative (symmetrical) behaviors and thus the design of such methods needs more elaboration.

In order to successfully exploit symmetries it is required to (1) define in a generic way, at the conceptual level, what method can be used to reduce the state space through symmetries (2) select a formalism where symmetries are automatically detected (3) define how the method can be efficiently implemented in practice. The design at the conceptual level is based on the operations of a permutation group; the formalism must allow a simple way to express similar components; the implementation should be based on a symbolic representation of set of states and transitions and their efficient manipulation.

However, the systems seldom have completely symmetric behavior (for example in distributed algorithms we often have a symmetric specification, together with some symmetry-breaking criteria - e.g. based on unique process identity - to solve conflicts, deadlocks, etc.) so it is useful to define and implement methods to deal with partial symmetries. In the literature partial symmetry methods have been proposed for qualitative analysis [8, 9, 10, 11].

In this paper we propose two methods to apply lumping in partially symmetrical models: they are here presented for the first time as *generic* methods, applicable to any kind of formalism that may originate partially symmetric models, while they were originally proposed in [12, 13] in the specific context of the Stochastic Well-Formed Net (SWN) formalism [5]. The first one, called Dynamic Symmetry (DS) method, starts from the synchronized product of a completely symmetric Markov chain (MC) and an additional automata describing the asymmetries: in this case a lumped MC satisfying the exact lumpability condition is built. The second one, called Two-Levels Symmetry (TLS) method, instead starts from an over-aggregated MC from which a lumped MC can be derived by applying a refinement algorithm: it can use either the strong lumpability or the exact lumpability condition (which have different impact on the type of performance indices that can be computed and may lead to different degrees of aggregation).

In the paper we show how the two methods can be efficiently applied to SWN[1] models, in fact this formalism is designed so that symmetries can be automatically detected and exploited. However since they are here presented as *generic* methods, it is also possible to adapt them to other kinds of high level stochastic models (as we show for two examples, Stochastic Automata Networks and Stochastic Activity Networks, discussed towards the end of the paper). One of the main differences between the two methods is that the TLS method uses two different aggregation criteria depending on the current phase of the behavior (symmetric or asymmetric), while the DS method aggregates states in a more dynamic way, possibly using several different aggregation criteria which may correspond to a more articulated classification of the behavior phases: symmetric behavior or one among several asymmetric behaviors.

We have implemented our methods in the GreatSPN tool [14] allowing us to perform several experiments. Six case studies are presented in the paper by means of the SWN formalism. The experimental results show that relevant savings in the state space size can be achieved through both approaches, and that they can be alternatively applied in the most appropriate situations.

The paper unifies and extends the results presented in [12, 13], revisiting them in a more general setting and giving a particular emphasis to the case studies. It is organized as follows: in Sec. 2 some basic notions on MC and lumpability are defined, in Sec. 3 the TLS and DS

[1]SWNs are high level stochastic Petri nets with a specific syntax for expressing color domains of places and transitions, arc functions and transition guards, that support automatic symmetry exploitation.

methods are presented; in Sec. 4 we show how they have been instantiated in the SWN formalism and we discuss about the implementation in GreatSPN. Sec. 5 describes which kind of partially symmetrical systems leads more effective state space reduction. In Sec. 6 significant case studies are presented and analyzed. Finally in Sec. 7 we give some hints on possible efficient application to other formalisms. We conclude in Sec. 8.

2. Markov chain lumpability

2.1. Strong, weak and exact lumpability

The quantitative evaluation of dynamic systems proposed in this paper implies the following three steps (a) the specification of the stochastic process representing the target system, (b) the definition of the required performance (or dependability) indices and (c) the possibility of applying efficient algorithms for transient or steady state measures computation.

The analysis of stochastic processes in general is a hard problem, in fact often simulation is the only viable option, while in some cases algorithms for the computation of approximations or bounds on the desired measures can be applied.

When the dynamic system behavior can be described through a finite Discrete Time MC (DTMC) or Continuous Time MC (CTMC), the solution is conceptually simpler, however, in realistic case studies, it is still computationally expensive; for this reason state space reduction techniques have been studied, such as the so called MC *lumping* technique [2].

Lumping of (finite) MCs is a useful method for dealing with large chains [17]. The principle is simple: substitute to the MC an "equivalent" one, where each state of the lumped chain is a set of states of the original one. There are different versions of lumpability related to the fact that the lumpability condition holds for every initial distribution (*strong lumpability*) or for at least one (*weak lumpability*). First, we briefly introduce MCs. Due to space constraints, we only deal with CTMCs. However our methods also apply to DTMCs and we indicate later on the interest of dealing with DTMCs even in a continuous time setting.

Definition 1 (Markov Chains). *A CTMC $\mathcal{C} = \langle S, Q, \pi_0 \rangle$ is defined by a state space S, an infinitesimal generator Q that is a $S \times S$ matrix whose off-diagonal elements are non negative reals, while each diagonal element is defined as $Q[s,s] = -\sum_{s \neq s'} Q[s,s']$, and π_0, an initial probability distribution over S. We note $\{X_t\}_{t \in \mathbb{R}_{\geq 0}}$ the associated stochastic process.*

Notation. S_0 denotes the subset of "initial" states, i.e., $S_0 = \{s \in S \mid \pi_0(s) > 0\}$.

We now introduce lumpability concepts.

Definition 2. *Let $\mathcal{C}$ be a CTMC and $\{S_i\}_{i \in I}$ be a partition of the state space. Let Y_t be a random variable defined by $Y_t = i \Leftrightarrow X_t \in S_i$. Then:*

- *Q is* strongly lumpable *w.r.t. $\{S_i\}_{i \in I}$ iff $\forall \pi_0$, $\{Y_t\}_{t \in \mathbb{R}_{\geq 0}}$ is a CTMC,*
- *Q is* weakly lumpable *w.r.t. $\{S_i\}_{i \in I}$ iff $\exists \pi_0$ s.t. $\{Y_t\}_{t \in \mathbb{R}_{\geq 0}}$ is a CTMC.*

[2] When a partition of states satisfies the lumpability condition it is possible to perform the aggregation of states efficiently without introducing approximations as it happens with other methods like e.g. [15, 16]

Whereas the characterization of strong lumpability w.r.t. the infinitesimal generator is straightforward, checking for weak lumpability is much harder [18]. Here, we introduce the exact lumpability, a simpler case of weak lumpability.

Definition 3. *Let $\mathcal{C}$ be a CTMC and $\{S_i\}_{i\in I}$ be a partition of the state space. Let Y_t be a random variable defined by $Y_t = i \Leftrightarrow X_t \in S_i$. Then:*
- *An initial distribution π_0 is equiprobable w.r.t. $\{S_i\}_{i\in I}$ if $\forall i \in I, \forall s, s' \in S_i, \pi_0(s) = \pi_0(s')$.*
- *Q is* exactly lumpable *w.r.t. $\{S_i\}_{i\in I}$ iff $\forall \pi_0$ equiprobable w.r.t. $\{S_i\}_{i\in I}$ $\{Y_t\}_{t\in \mathbb{R}_{\geq 0}}$ is a CTMC.*

Exact and strong lumpability have easy characterizations [19] given by the following proposition.

Proposition 4. *Let $\mathcal{C}$ be a CTMC and $\{S_i\}_{i\in I}$ be a partition of the state space. Then:*
- *Q is strongly lumpable w.r.t. $\{S_i\}_{i\in I}$ iff $\forall i \neq j \in I$, $\forall s, s' \in S_i$, $\sum_{s''\in S_j} Q(s, s'') = \sum_{s''\in S_j} Q(s', s'')$,*
- *Q is exactly lumpable w.r.t. $\{S_i\}_{i\in I}$ iff $\forall i, j \in I$, $\forall s, s' \in S_i$, $\sum_{s''\in S_j} Q(s'', s) = \sum_{s''\in S_j} Q(s'', s')$.*

The following corollary establishes a sufficient condition for exact lumpability in CTMCs which will be useful in order to check the correctness of one of our methods.

Corollary 5. *Let $\mathcal{C}$ be a CTMC and $\{S_i\}_{i\in I}$ be a partition of the state space. Then Q is exactly lumpable w.r.t. $\{S_i\}_{i\in I}$ if:*
1. $\forall i \neq j \in I, \forall s, s' \in S_i$, $\sum_{s''\in S_j} Q(s'', s) = \sum_{s''\in S_j} Q(s'', s')$.
2. $\forall i \in I, \forall s, s' \in S_i$, $\sum_{s''\neq s\in S_i} Q(s'', s) = \sum_{s''\neq s'\in S_i} Q(s'', s')$.
3. $\forall i \in I, \forall s, s' \in S_i, Q(s, s) = Q(s', s')$.

When the strong lumpability condition holds the infinitesimal generator of the lumped chain can be directly computed from the original generator as expressed by the following proposition.

Proposition 6. *Let $\mathcal{C}$ be a CTMC that is strongly lumpable w.r.t. a partition of the state space $\{S_i\}_{i\in I}$. Let Q^{lp} be the generator associated with this lumped CTMC, then:*

$$\forall i, j \in I, \forall s \in S_i, Q^{lp}(i, j) = \sum_{s'\in S_j} Q(s, s').$$

As for strong lumpability, also in case of exact lumpability the infinitesimal generator of the lumped chain can be directly computed from the original generator. Observe that starting with the probability mass equidistributed on the states of every subset of the partition, the distribution at any time is still equidistributed. Consequently, if the CTMC is ergodic, its steady-state distribution is equidistributed between states of every subset of the partition. In other words, with the knowledge of the lumped chain generator, one may compute its steady-state distribution, and deduce (by *local* equidistribution) the steady-state distribution of the original chain. It must be emphasized that this last step is impossible with strong lumpability since it does not ensure equiprobability of the states in an aggregate.

Proposition 7. *Let $\mathcal{C}$ be a CTMC that is exactly lumpable w.r.t. a partition of the state space $\{S_i\}_{i\in I}$. Let Q^{lp} be the generator associated with this lumped CTMC, then:*

- $\forall i, j \in I, \forall s \in S_j$,
  $Q^{lp}(i,j) = (\sum_{s'\in S_i} Q(s', s)) \times (|S_j|/|S_i|)$
- *If* $\forall i \in I, \forall s, s' \in S_i, \pi_0(s) = \pi_0(s')$ *then* $\forall t \in \mathbb{R}_{\geq 0}$,
  $\forall i \in I, \forall s, s' \in S_i, \pi_t(s) = \pi_t(s')$,
  where π_t is the probability distribution at time t.
- *If Q is ergodic and π is its steady-state distribution*
  then $\forall i \in I, \forall s, s' \in S_i, \pi(s) = \pi(s')$.

Observation: the above aggregation equations for strong and exact lumpability can be derived from the general aggregation equation $Q^{lp}(i,j) = \frac{\sum_{s'\in S_i} \pi(s') \sum_{s\in S_j} Q(s',s)}{\sum_{s'\in S_i} \pi(s')}$ applying the equations in Proposition 4 and the fact that when exact lumpability holds the steady state distribution π is equiprobable w.r.t. $\{S_i\}_{i\in I}$.

2.2. Dealing with DTMCs.

As said before, very similar results hold for DTMCs. Furthermore even in a continuous time setting, there are two situations where choosing DTMCs is convenient. Some semi-Markovian processes are analyzable *via* an *embedded* DTMC which only takes into account state changes. This DTMC could be lumpable thus enlarging this technique to semi-Markovian processes. Furthermore, it may happen that even in a case of CTMC, the embedded DTMC has a greater reduction factor by lumpability. We have experienced this phenomenon when benchmarking our methods.

2.3. Computation of performance indices.

Let us now recall how it is possible to characterize the performance index (or indices) of interest on a given CTMC, and then discuss the implications of lumping on its computability.

The performance indices of interest can be computed in a transient or steady state setting. Examples of performance indices are the steady state availability of a server, the probability that a given connection be active at time instant τ, or the average number of clients being served in a system.

A general way of defining performance indices on CTMCs is through the use of *reward functions*: their domain is the set S of CTMC states while the co-domain is $\mathbb{R}$. In fact, a function r can be seen as a performance index and, given a (steady state or transient) state probability distribution π, the (average or instantaneous) performance index measure can be expressed as: $\sum_{s\in S} \pi(s) \cdot r(s)$.

If the reward function r expressing the performance index of interest is constant within each aggregate, then the probability distribution of the aggregates is enough to compute the value of the performance index (we can say that the reward function is compatible with the aggregation). However if this is not the case, only exact lumpability still gives us the possibility to compute the performance index value.

Finally observe that the efficient computation of performance indices corresponding to unconstrained reward functions in the exact lumpability case requires a way of efficiently computing the

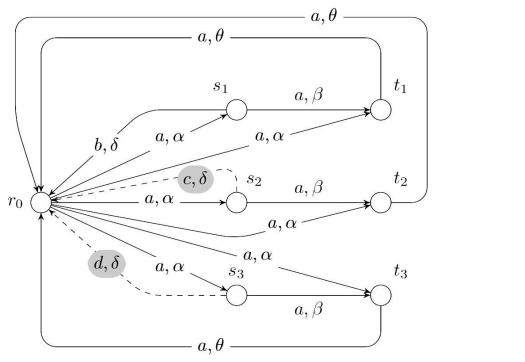
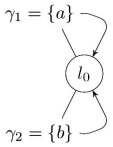


Figure 1: A labeled CTMC and its control automaton



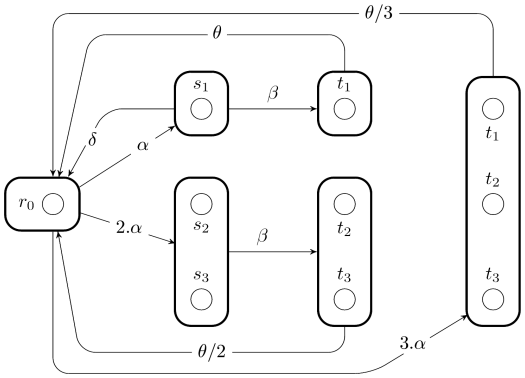


Figure 2: CTMC $(\mathcal{C}_{\mathcal{A}}^{G})^{lp}$

cardinality of each aggregate, and of the subset of states within the aggregate characterized by the same reward function value.

3. The DS and TLS methods

3.1. Lumpability of partially symmetrical MCs

This section presents the *Dynamic Symmetry* method applied to partially symmetrical MCs.

Partially symmetrical CTMCs. The model of partially symmetrical systems that we develop here is defined as a CTMC obtained by some synchronized product between a (symmetrical) CTMC and a control automaton. Let us first formalize this product. Synchronizing the behavior of the two components requires to "label" the CTMC with events.

Notation. Let $\mathcal{C}$ be a CTMC, we associate with each pair of states $s \neq s'$ a label in some alphabet $\Sigma \cup \{\varepsilon\}$, denoted $\Lambda(s, s')$. We require that $\Lambda(s, s') = \varepsilon$ iff $Q(s, s') = 0$.

Since the automaton is introduced in order to modify the behavior of the CTMC, the label of each edge is a predicate that selects the events allowed to occur in the current location of the automaton.

Definition 8. *Let $\mathcal{C}$ be a CTMC, then $\mathcal{A} = \langle L, l_0, \rightarrow \rangle$ a control automaton of $\mathcal{C}$ is defined by:*

- *L, the set of automaton locations,*
- *l_0, the initial location,*
- *$\rightarrow \subseteq L \times 2^{\Sigma} \times L$, the transitions of the automaton.*
  A transition (l, γ, l') will be denoted by $l \xrightarrow{\gamma} l'$.
  Furthermore, if $l \xrightarrow{\gamma} l'$ and $l \xrightarrow{\gamma'} l'$ with $\gamma \neq \gamma'$
  then $\gamma \cap \gamma' = \emptyset$.

In standard automata, the last requirement can be easily ensured by merging the two transitions into a single one labeled by $\gamma \cup \gamma'$. However the interest of letting distinct the two transitions will be discussed later.

Fig.1 represents a CTMC and its control automaton. Standard letters are labels, while Greek letters represent transition rates. The initial distribution is: $\pi(r_0) = 1$.

In the synchronized product defined below, the CTMC is the "active" component whereas the automaton is the "passive" component waiting for a transition of the CTMC in order to synchronize

it with one of its transitions. Consequently, the rates (resp. the initial distribution) associated with the product depends only on the rates (resp. the initial distribution) of the CTMC.

Definition 9. *Let $\mathcal{C}$ be a CTMC and $\mathcal{A}$ some control automaton of $\mathcal{C}$. The synchronized product of $\mathcal{C}$ and $\mathcal{A}$, $\mathcal{C}_{\mathcal{A}} = \langle S \times L, \pi'_0, Q' \rangle$ is a CTMC defined by:*

- $\forall s, \pi'_0(s, l_0) = \pi_0(s) \wedge \forall l \neq l_0, \pi'_0(s, l) = 0$
- $\forall s \neq s' \in S, \forall l, l' \in L$, *if* $l \xrightarrow{\gamma} l' \wedge \Lambda(s, s') \in \gamma$
  then $Q'((s, l), (s', l')) = Q(s, s')$
  else $Q'((s, l), (s', l')) = 0$
- $\forall s \in S, \forall l \neq l' \in L$, $Q'((s, l), (s, l')) = 0$

Remarks. Due to the constraint on the labeling function Λ, a transition with null rate cannot be synchronized with an automaton transition. The requirement related to transitions of the control automaton ensures that given a current location l, a possible next location l' and a label $\alpha \in \Sigma$ (triggered by a transition of the CTMC) there is at most one transition of the automaton that reaches l' from l accepting label α. In realistic applications, the control automaton is only used in order to restrict the behavior of the original CTMC. However observe that the outgoing transition rate of a state (s, l) can be greater than the one of s. Take for instance $\Lambda(s, s') = \alpha$, $l \xrightarrow{\{\alpha\}} l'$ and $l \xrightarrow{\{\alpha\}} l''$ and assume that $Q(s, s) = -Q(s, s')$ (i.e., s' is the only successor of s). Then $Q((s, l), (s, l)) = 2Q(s, s)$ due to the two automaton arcs. We choose this more general setting since for specific applications, it could be useful.

In the example of Fig.1, the control automaton actually forbids transitions that are not labeled with a or b. Hence, $\mathcal{C}_{\mathcal{A}}$ is obtained from $\mathcal{C}$ by removing the dotted arcs. Formally, the states of $\mathcal{C}_{\mathcal{A}}$ are pairs (s_i, l) but as there is only one location in the automaton, we will omit it in the representation of states throughout the example.

From a theoretical point of view, the specification of the system symmetries relies on group theory, applied to the states and the events of the system. The next definition recalls the appropriate notions.

Definition 10. *Let G be a group, with neutral element id and whose internal operation is denoted ($\bullet$). Let E be a set.*

- *An* operation *of G on E is a mapping from $G \times E$ to E s.t. the image of (g, e), denoted by $g.e$, fulfills:*
  $\forall e \in E, id.e = e \wedge \forall g, g' \in G, (g \bullet g').e = g.(g'.e)$
- *The* isotropy subgroup *of a subset $E' \subseteq E$ is defined by:*
  $G_{E'} = \{g \in G \mid \forall e \in E', g.e \in E'\}$
- *Let H be a subgroup of G, the* orbit *of e by H denoted $H.e$, is defined by: $\{g.e \mid g \in H\}$.*
  The set of orbits by H defines a partition of E.

We simultaneously introduce the notions of symmetrical and partially symmetrical CTMCs. Informally, a CTMC is *symmetrical* w.r.t. some group if the operation of the group on the state

space preserves its initial distribution and stochastic behavior. A CTMC is *partially symmetrical* if it is a synchronized product of a symmetrical CTMC with a (non symmetrical) control automaton.

Definition 11. *A CTMC $\mathcal{C}$ is symmetrical w.r.t. a group G operating on S and Σ iff:* $\forall g \in G, \forall s \neq s' \in S, \pi_0(g.s) = \pi_0(s) \wedge Q(g.s, g.s') = Q(s, s')$ *and* $\Lambda(g.s, g.s') = g.\Lambda(s, s')$.

Let $\mathcal{C}$ be symmetrical w.r.t. G and $\mathcal{A}$ be a control automaton of $\mathcal{C}$, then $\mathcal{C}_\mathcal{A}$ is said to be partially symmetrical w.r.t. G.

We associate with each γ occurring in a transition of $\mathcal{A}$ a subgroup $H_\gamma \subseteq G$ defined by: $g \in H_\gamma$ iff $\forall a \in \Sigma, a \in \gamma \Leftrightarrow g.a \in \gamma$.

The size of the subgroup H_γ is an indicator of the symmetry of the associated edge. When $H_\gamma = G$, the edge is "fully" symmetrical whilst when $H_\gamma = \{id\}$, the edge is "fully" asymmetrical. Here we see the interest of keeping distinct transitions of the control automaton with same sources and destinations. Indeed when merging them, the subgroup associated with the new transition could be smaller than one of (or even both) the subgroups associated with the original transitions.

Back to the example of Fig.1, let G be the group of permutations of $\{1, 2, 3\}$ generated by binary permutations $p_{i,j}$ which exchange i and j. The operations of G on S and Σ are defined by:
$\forall p_{i,j}, p_{i,j}.r_0 = r_0 \wedge p_{i,j}.a = a$
$\forall p_{i,j}, p_{i,j}.s_i = s_j \wedge p_{i,j}.t_i = t_j$
$\forall p_{i,j}, p_{i,j}.s_j = s_i \wedge p_{i,j}.t_j = t_i$
$\forall p_{i,j}, k \notin \{i, j\}, p_{i,j}.s_k = s_k \wedge p_{i,j}.t_k = t_k$
$p_{1,2}.b = c \wedge p_{1,2}.c = b \wedge p_{1,2}.d = d$
$p_{1,3}.b = d \wedge p_{1,3}.c = c \wedge p_{1,3}.d = b$
$p_{2,3}.b = b \wedge p_{2,3}.c = d \wedge p_{2,3}.d = c$

It is easy to verify that the CTMC is symmetrical w.r.t. G. The subgroups associated with the labels of $\mathcal{A}$ are $H_{\gamma_1} = G$ and $H_{\gamma_2} = \{id, p_{2,3}\}$. Observe that if instead we had merged the transitions, the group would have been $\{id, p_{2,3}\}$ and thus the full symmetry of the edge γ_1 would have been lost.

A subset construction for lumpability. Given a partially symmetrical CTMC $\mathcal{C}_\mathcal{A}$, our method builds a smaller (but equivalent) CTMC based on the building of some "subset" reachability graph that we call $\mathcal{G}_\mathcal{A}$. Algorithm 1 describes its construction.

Let us detail how it works. The nodes of this graph are pairs consisting in a location of $\mathcal{A}$ and a subset of states of $\mathcal{C}$ which equivalently denotes a subset of states of $\mathcal{C}_\mathcal{A}$ with same location. An edge of this graph is labeled by a transition $l \xrightarrow{\gamma} l'$ of $\mathcal{A}$ and it represents a (non empty) set of transitions of $\mathcal{C}_\mathcal{A}$. More precisely, such a transition links some state of the source subset to some state of the destination subset that can be reached using $l \xrightarrow{\gamma} l'$.

The key idea of this construction is the following: along any path of this graph (and independently on the instants of transition firings corresponding to the arcs of this path) starting from the initial distribution, the occurrence probability of all states of the subset associated with the last node of this path are identical.

In fact, the construction maintains the following invariants: (1) The graph represents all behaviors except possibly the ones that start from some destination node of an edge that is present in the

Algorithm 1: Building of $\mathcal{G}_{\mathcal{A}}$

1 $nodes = \emptyset$; $edges = \emptyset$;
2 Partition $S_0 = \uplus_{i=1}^{n_0} S_{0,i}$
3 s.t. every $S_{0,i}$ is the orbit of some $s_i \in S_0$ by G;
4 add $\perp$ to *nodes*;
5 **for** $i \in \{1, \ldots, n_0\}$ **do**
6 $\quad push(stack, \perp \xrightarrow{init} (l_0, S_{0,i}))$;
7 **while** *stack is not empty* **do**
8 $\quad (l, R) \xrightarrow{\gamma} (l', R') = pop(stack)$;
9 $\quad$ Compute $\Gamma = \{\gamma' \mid \exists l' \xrightarrow{\gamma'} l'', \exists s' \in R', \exists s'' \in S, \Lambda(s', s'') \in \gamma'\}$;
10 $\quad$ Compute $H = G_{R'} \cap \bigcap_{\gamma' \in \Gamma} H_{\gamma'}$;
11 $\quad$ Partition $R' = \uplus_{i=1}^{m} R_i$ s.t. every R_i is the orbit of some $r_i \in R'$ by H;
12 $\quad$ **for** $i \in \{1, \ldots, m\}$ **do**
13 $\quad\quad$ **if** $(l', R_i) \in nodes$ **then**
14 $\quad\quad\quad add(l, R) \xrightarrow{\gamma} (l', R_i)$ to *edges*;
15 $\quad\quad$ **else**
16 $\quad\quad\quad add(l', R_i)$ to *nodes*;
17 $\quad\quad\quad add(l, R) \xrightarrow{\gamma} (l', R_i)$ to *edges*;
18 $\quad\quad\quad$ **for** $l' \xrightarrow{\gamma'} l''$ **do**
19 $\quad\quad\quad\quad$ Compute $\mathcal{SETS} = \{H.s^* \mid \Lambda(r_i, s^*) \in \gamma'\}$;
20 $\quad\quad\quad\quad$ **for** $S' \in \mathcal{SETS}$ **do**
21 $\quad\quad\quad\quad\quad push(stack, (l', R_i) \xrightarrow{\gamma'} (l'', S'))$;

stack. (2) The nodes (i.e., the corresponding subset of states) of the graph fulfill all the conditions of corollary 5. (3) The subsets which are destination of an edge in the stack fulfill the first two conditions of corollary 5.

The **while** loop extracts an edge from the stack (line 8). Then it splits the destination subset R' (lines 9-11) in order to ensure the third condition of corollary 5 since inside a subset R_i, the states allow the same transitions of the control automaton. Furthermore, $r_i \in R_i$ is selected. If R_i is a node of the graph (lines 13-14) then one adds the edge to the graph (while preserving the conditions of corollary 5). Otherwise one creates R_i as a new node and the corresponding incoming edge and computes the outgoing edges of R_i (lines 18-24). The variable $\mathcal{SETS}$ contains orbits w.r.t. H reachable from R_i using a transition whose label belongs to γ'. These edges are pushed onto the stack. Again by construction, the destination subsets of states fulfill the first two conditions of corollary 5. Furthermore, the choice of the r_i (line 19) is irrelevant since R_i is the orbit under H of any of its item. So whatever the choice, the set of subsets $\mathcal{SETS}$ will be identical.

The initial stage consists in partitioning the initial states (S_0) w.r.t. G (line 2). Since there is no incoming edge the two first conditions of corollary 5 are satisfied. We have added a fictitious node $\perp$ in order to handle the $S_{0,i}$ subsets in the main loop (lines 3-6).

We have not represented the computation of rates in the algorithm in order to focus on the qualitative aspects. We explain it now: Let $(l, R) \xrightarrow{\gamma} (l', R_i)$ be an edge of $\mathcal{G}_{\mathcal{A}}$. We apply proposi-

tion 7; so we select a state $s' \in R_i$ and compute $(\sum_{s' \in R} Q(s', s)) \times (|R_i|/|R|)$. The first term only depends on s and so it is efficiently computed. Varying with the specification formalisms, the computation of cardinalities can be done more or less efficiently but it is always a local computation and then it does not suffer a combinatorial explosion. Furthermore some tricks are possible. For instance, R is the orbit of any state $r \in R$ by some group say H. So we have $|R| = |H|/|H \cap G_r|$ (by an elementary result on groups). With some formalisms the computations of cardinalities of such groups is straightforward and leading to efficiently obtain $|R|$.

In order to prove the soundness of this construction, we first introduce a CTMC $\mathcal{C}_{\mathcal{A}}^{G}$, which is bigger than $\mathcal{C}_{\mathcal{A}}$. In $\mathcal{C}_{\mathcal{A}}^{G}$, states of $\mathcal{C}_{\mathcal{A}}$ are replicated in instances, and instances are organized w.r.t. the subsets associated with the nodes of $\mathcal{G}_{\mathcal{A}}$. By construction, all the instances that belong to the same subset have the same associated location of the automaton. We will denote (s, l, R) the instance of (s, l) s.t. s belongs to such a subset R. In the next definition, *nodes* (resp. *edges*) refers to the nodes (resp. edges) of $\mathcal{G}_{\mathcal{A}}$.

Definition 12. *Let $\mathcal{C}_{\mathcal{A}}$ be partially symmetrical CTMC w.r.t. G, then the CTMC $\mathcal{C}_{\mathcal{A}}^{G} = \langle S'', \pi_0'', Q'' \rangle$ is defined by:*

- *The set of states S'' is defined by:*
  $S'' = \{(s, l, R) \mid (l, R) \in nodes \wedge s \in R\}$.
- $\forall i \in \{1, \ldots, n_0\}, \forall R$ *s.t. R is an item of the partition of $S_{0,i}$,* $\forall s \in R$, $\pi_0''(s, l_0, R) = \pi_0'(s, l_0)(= \pi_0(s))$.
  For every other $(s, l, R) \in S''$, $\pi_0''(s, l, R) = 0$.
- $\forall (s, l, R) \neq (s', l', R') \in S''$, *If* $(l, R) \xrightarrow{\gamma} (l', R')$ *is in* $edges$ *then* $Q''((s, l, R), (s', l', R')) = Q(s, s')$.
  Otherwise $Q''((s, l, R), (s', l', R')) = 0$.

The stochastic process we want to build is obtained by forgetting the instances and only memorizing the subsets.

Definition 13. *Let $\mathcal{C}_{\mathcal{A}}$ be partially symmetrical w.r.t. G, then the stochastic process $(\mathcal{C}_{\mathcal{A}}^{G})^{lp}$ is defined by: $X_t^{lp} = (R, l)$ iff $X_t'' \in \{(s, l, R)\}$.*

The next proposition is the theoretical core of our method. It states that $(\mathcal{C}_{\mathcal{A}}^{G})^{lp}$ is obtained from $\mathcal{C}_{\mathcal{A}}$ by the inverse of a strong followed by an exact lumping.

Proposition 14. *Let $\mathcal{C}_{\mathcal{A}}$ be partially symmetrical w.r.t. G, then:*

- *Denoting $(s_0, l_0) \ldots, (s_n, l_n)$ the state space of $\mathcal{C}_{\mathcal{A}}$, $\mathcal{C}_{\mathcal{A}}$ is a strong lumping of $\mathcal{C}_{\mathcal{A}}^{G}$ w.r.t. the partition $\biguplus sl_i$ where $sl_i = \{(s_i, l_i, R) \in S''\}$.*
- *Denoting $\{(R_0, l_0), \ldots, (R_k, l_k)\}$ the state space of $(\mathcal{C}_{\mathcal{A}}^{G})^{lp}$, $(\mathcal{C}_{\mathcal{A}}^{G})^{lp}$ is an exact lumping of $\mathcal{C}_{\mathcal{A}}^{G}$ w.r.t. the partition $\biguplus Rl_i$ where $Rl_i = \{(s, l_i, R_i) \in S''\}$.*

Proof

Let (s, l) be a state of $\mathcal{C}_{\mathcal{A}}$ and let (s, l, R) be an instance of this state in $\mathcal{C}_{\mathcal{A}}^{G}$, we show that there

is a bijective mapping from the transitions out of (s, l) onto the transitions out of (s, l, R). So we can suppose that s is examined when looking for successors of (l, R). Then $\exists s', \exists l \xrightarrow{\gamma} l'$ s.t. $\Lambda(s, s') \in \gamma \Leftrightarrow \exists R', \exists s' \in R', \exists l \xrightarrow{\gamma} l'$ s.t. $\Lambda(s, s') \in \gamma$ with (l', R') a successor of (l, R). Since this mapping preserves the rate of the transitions the condition of Prop. 4 for strong lumpability is fulfilled.

Let (s_1, l, R) and (s_2, l, R) be two states of $\mathcal{C}_{\mathcal{A}}^G$, we show that there is a bijective mapping from the input transitions of (s_1, l, R) onto the input transitions of (s_2, l, R). Let (v_1, l', R') be such that $\exists l' \xrightarrow{\gamma} l$ and $\Lambda(v_1, s_1) \in \gamma$. Let H be the group of line 10 related to l', R', then $\exists g \in H \subseteq G_{R'} \cap H_\gamma$ s.t. $s_2 = g.s_1$. Now define $v_2 = g.v_1$, then $v_2 \in R'$ and $\Lambda(v_2, s_2) \in \gamma$. This implies the existence of the required mapping. Since this mapping preserves the rates of transitions, the first two conditions of corollary 5 for exact lumpability are fulfilled. The third one is ensured by the splitting of line 11 which has produced (l, R). □

Illustration. We illustrate the algorithm on the CTMC of Fig. 1. The lumped CTMC $(C_{\mathcal{A}}^G)^{lp}$ is given in Fig. 2. We have represented inside each node the states corresponding to the subset associated with that node. Let us describe the first steps of the algorithm. We push on the stack the edge $\perp \xrightarrow{init} (l_0, \{r_0\})$. When we pick it, we determine that only the automaton transition labeled by a can be synchronized. Thus the subgroup of line 10, H is equal to G.

The transition $(r_0, l_0) \xrightarrow{a} (s_1, l_0)$ (resp. $(r_0, l_0) \xrightarrow{a} (t_1, l_0)$) yields to push on the stack an edge whose destination set is $\{s_1, s_2, s_3\}$ (resp. $\{t_1, t_2, t_3\}$). When the edge with destination $\{s_1, s_2, s_3\}$ is popped, the two transitions of the automaton can be synchronized and thus the group H of line 10 becomes $\{id, p_{2,3}\}$. The orbits of $\{s_1, s_2, s_3\}$ w.r.t. H are $\{s_1\}$ and $\{s_2, s_3\}$. At the end, observe that states t_i appear twice: in $\{t_1, t_2, t_3\}$ and in some orbit of $\{id, p_{2,3}\}$. We can intuitively explain it as follows. When the CTMC reaches directly the states t_i from r_0 then their occurrence is equiprobable which is only the case for t_2 and t_3 when going through s_i.

Our generic method can now be described. Assume first that the CTMC $\mathcal{C}_{\mathcal{A}}$ associated with the high-level model $\mathcal{M}$ we want to analyze is partially symmetrical. Assume also that we are able to compute directly $(\mathcal{C}_{\mathcal{A}}^G)^{lp}$ from $\mathcal{M}$. Note π_t the unknown distribution of $\mathcal{C}_{\mathcal{A}}$ at time t and $\pi_t^{(lp)}$ the (computed) distribution of $(\mathcal{C}_{\mathcal{A}}^G)^{lp}$ at time t. Then $\pi_t(s, l) = \sum_{s \in R} (1/|R|) \times \pi_t^{(lp)}(R, l)$. The equality also holds for the steady-state distributions.

Although theoretically difficult, we can give some hints of how the space complexity decreases using our approach. In the lumped CTMC, the original states have been substituted by subsets. Note that these subsets may intersect. However these subsets are always the orbit of a state by a subgroup of G. Thus, the larger these subgroups, the better the method. Note that each time a new subset is built, the group is reduced (by intersection with some groups H_γ) and then is enlarged by implicitly substituting to these intersections, the isotropy subgroup of the subset. Interpreting this phenomenon at the model level, we deduce that the complexity reduction factor is high whenever the effect of an asymmetrical event is forgotten in a close future. Experimentations will illustrate this interpretation.

3.2. Lumpability of Almost Symmetrical MCs

In this section we shall define the second method for the (strong or exact) lumpability of a finite CTMC, called *Two-Levels Symmetry* (TLS) method.

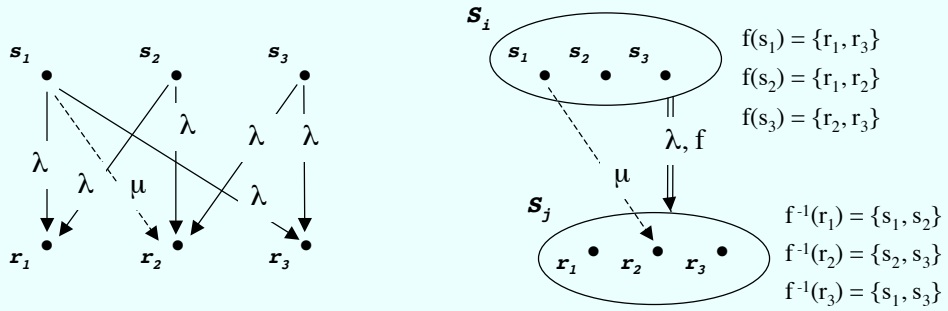


Figure 3: A simple CTMC and its almost symmetrical specification

The starting point is an *almost symmetrical specification* of the CTMC. This specification is a graph, whose nodes are aggregates of a partition of CTMC states. These nodes are connected by two types of arcs: *generic* and *instantiated* arcs. Generic arcs result from the presence of symmetries in the system: the actions that they represent are performed similarly from all the states belonging to the same partition aggregate. Hence, these arcs can be defined at the aggregate level. On the contrary, *instantiated* arcs result from asymmetry and represent actions that are performed individually. A simple example of almost symmetrical specification of a CTMC is presented in Fig. 3.

Such arc specification leads to a compact representation of the CTMC; unfortunately this representation may not not satisfy any lumpability condition due to the presence of instantiated arcs. Hence, the idea is to derive from this structure a new one for which lumpability holds. To do so a partition refinement algorithm is applied.

Definition 15 (Almost symmetrical CTMC). *An almost symmetrical specification of a CTMC* $\mathcal{C} = \langle S, \pi_0, Q \rangle$ *is defined by:*

- *a partition of the state space* $\{S_i\}_{i \in I}$ *such that* $S = \uplus_{i \in I} \{S_i\}$
- *for* $S_i, S_j \in \{S_i\}_{i \in I}$*, two types of state transition arcs:*
  generic arcs*:* $S_i \xrightarrow{\lambda, f} S_j$ *where* $\lambda \in \mathbb{R}_{>0}$ *is a rate, and* f *is a function* $f : S_i \to 2^{S_j}$*, such that*

  0. $\forall s \in S_i \ : \ r \in f(s) \Rightarrow Q[s, r] \geq \lambda$,

  1. $\forall s, s' \in S_i, |f(s)| = |f(s')|$,

  2. $\forall r, r' \in S_j, |f^{-1}(r)| = |f^{-1}(r')|$,
  where f^{-1} *is the function* $S_j \to 2^{S_i}$ *defined by* $f^{-1}(r) = \{s \mid r \in f(s)\}$.

  instantiated arcs*:* $s \xrightarrow{\mu} r$ *where* $s \in S_i$*,* $r \in S_j$ *and* $\mu \in \mathbb{R}_{>0}$ *is a rate, such that :*

$$Q[s, r] = \sum_{S_i \xrightarrow{\lambda, f} S_j, r \in f(s)} \lambda + \sum_{s \xrightarrow{\mu} r} \mu$$

Remark: Point (0.) of definition 15 relies on the fact that when the transition rate from s to r is greater than λ, it can always be decomposed into (at least) two arcs, one of which is labeled by λ.

A direct consequence of the definition is that, if the almost symmetrical specification of a CTMC is such that:

$$\forall S_i, S_j, \forall s, s' \in S_i, \sum_{r \in S_j} Q[s,r] = \sum_{r \in S_j} Q[s',r]$$

then Q is strongly lumpable with respect to $\{S_i\}_{i \in I}$. To ensure this property, it is sufficient that $\sum_{r \in S_j : s \xrightarrow{\mu} r} \mu = \sum_{r \in S_j : s' \xrightarrow{\mu'} r} \mu'$. In this case, the infinitesimal generator of the lumped CTMC is given by (see Proposition 6):

$$Q^{lp}(i,j) = \sum_{r \in S_j} Q[s,r] = \Big(\sum_{S_i \xrightarrow{\lambda, f} S_j} \lambda |f(s)| \Big) + \Big(\sum_{r \in S_j : s \xrightarrow{\mu} r} \mu \Big) \tag{1}$$

On the other hand, if the almost symmetrical specification of a CTMC is such that:

$$\forall S_i, S_j, \forall r, r' \in S_j, \sum_{s \in S_i} Q[s,r] = \sum_{s \in S_i} Q[s,r']$$

plus the following initial condition: $\forall i \in I, \forall s, s' \in S_i, \pi_0(s) = \pi_0(s')$, then Q is exactly lumpable with respect to $\{S_i\}_{i \in I}$. To ensure this property, it is sufficient that $\sum_{s \in S_i : s \xrightarrow{\mu} r} \mu = \sum_{s \in S_i : s \xrightarrow{\mu'} r'} \mu'$. In this case, the infinitesimal generator of the lumped CTMC is given by (see Proposition 7) :

$$Q^{lp}(i,j) = \frac{|S_j|}{|S_i|} \sum_{s \in S_i} Q[s,r] = \frac{|S_j|}{|S_i|} \Big(\Big(\sum_{S_i \xrightarrow{\lambda, f} S_j} \lambda |f^{-1}(r)| \Big) + \Big(\sum_{s \in S_i : s \xrightarrow{\mu} r} \mu \Big) \Big)$$

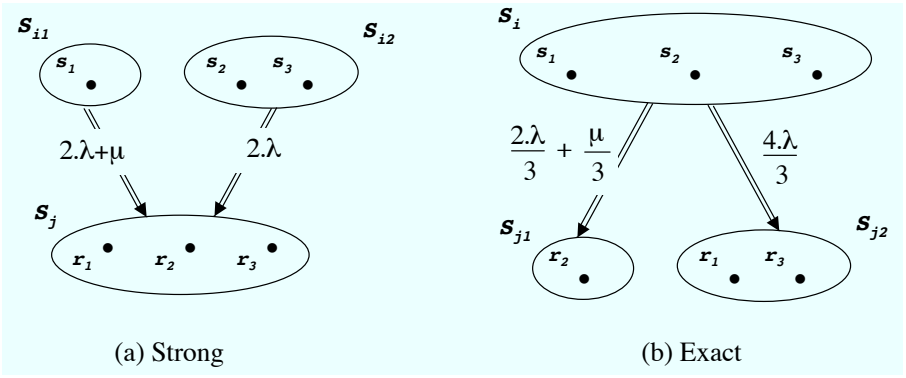
As $\forall s \in S_i, |f^{-1}(r)|.|S_j| = |f(s)|.|S_i|$, we obtain

$$Q^{lp}(i,j) = \sum_{S_i \xrightarrow{\lambda, f} S_j} \lambda |f(s)| + \frac{|S_j|}{|S_i|} \Big(\sum_{s \in S_i : s \xrightarrow{\mu} r} \mu \Big) \tag{2}$$

The above conditions for lumpability do not hold in general for an almost symmetrical CTMC specification, hence we propose an algorithm that iteratively refines the partition of the CTMC until the desired lumping condition is satisfied. Considering the example in Fig. 3, our algorithm will produce the structures in Fig 4, depending on whether we want to ensure (a) strong or (b) exact lumpability.

Our algorithm is based on (an adaptation of) Paige and Tarjan's partition refinement algorithm [20, 21, 22] and exploits the properties of generic arcs whenever possible to reduce the number of checks to be performed.

It is worth noting that the achieved lumpable CTMC can have more nodes than the one obtained with a coarser initial partition, however in the case where the initial aggregates cannot anyway be



(a) Strong (b) Exact

Figure 4: Refinement applied to the example of Fig. 3

Algorithm 2: Algorithm for the exact lumpability check

```
1  A, X : Set Of Sets of States (SSS);
2  B, D : Set Of States (SS);
3  Lel : Set of tuples ⟨real, integer, S⟩;
4  PartLel : Set of tuples ⟨SS, real, integer⟩;
5  X.Create(AS_CTMC);
6  A = X.PreSplit();
7  while X ≠ A do
8      D = X.Remove() s.t. ∀A_i ∈ A, A_i ≠ D;
9      B = A.Pick(D) s.t. ∃B ⊆ D ⇒ ∀A_i ⊆ D, |B| ⩾ |A_i|;
10     X.Insert(B);
11     X.Insert(D \ B);
12     Lel = CompAllSucc(B);
13     PartLel = Partition_wrt_rate_A(Lel);
14     A.Split(PartLel) ;
15 return A;
```

Algorithm 3: $SSSM$ :: $Split(PartLel)$

```
1  Set, A_i : SSM;
2  for ⟨S, rate, i⟩ ∈ PartLel do
3      Set = ∅;
4      A_i = GetElement(i);
5      Set = A_i \ S;
6      Substitute(i, Set);
7      Add(S);
```

lumped, then the number of steps of the present algorithm is less than the number of steps required when applying the algorithm directly on the original CTMC.

In practice, the choice of the initial partition is usually guided by the need to have an efficient (implicit and symbolic) representation of aggreates of the almost symmetrical CTMC. Moreover it can be related to the way performance indices are specified (e.g. through a reward function that is forced to have uniform value for all states within the same initial aggregate).

The efficiency of the proposed method relies on a compact (symbolic) representation of both generic arcs and partition aggregates. Of course a way of retrieving ordinary states and/or arcs must be given, as it is needed during the refinement.

The algorithm for checking exact lumpability A mapping between the *stability condition* of Paige and Tarjan's algorithm and the strong or exact lumpability condition is possible. This is easy for strong lumpability condition, since the stability condition is implied by it. In fact the stability condition requires that all elements in each aggregate reach the same set of destination

aggregates, while the strong lumpability condition also requires that they do so with the same rate.

Instead for the exact lumpability the mapping requires to consider the arcs as if they were reversed: when considering reversed arcs, again the stability condition is weaker than the exact lumpability condition. In fact the "reversed arcs" stability condition requires that all elements in each aggregate are reached by the same set of source aggregates while the exact lumpability condition also requires that they do so with the same rate; moreover we use the additional requirement that the global output rate of states in the same aggregate must be equal (Corollary 5 condition 3).

Like the Paige and Tarjan's Partition refinement algorithm, our algorithm uses the following data structures:
(1) A is the current partition of states[3]; every element of the list will be called *block*. A single block contains a set of elements of type *Node*. Moreover a *Node* can be a single state if it represents only one state; or "macrostate" if it represents an aggregate.
(2) X represents another possible partition into aggregates, such that A is a refinement of X and A satisfies the lumpability condition with respect to every block of X.

Algorithm 2 shows the pseudo-code of the algorithm. It has two main phases: the initialization (lines 5-6) and the iterative refinement (lines 7-14).

(1) The initial phase. $Create$ initializes the set X (of the set of states, hereafter called blocks) on the basis of the initial partition of the almost symmetrical CTMC specification: for each aggregate having only generic input and output arcs, a new block is inserted into X, containing only one element of type "macrostate". For each aggregate having also instantiated input and/or output arcs a new subset is inserted into X, containing as many elements of type "state" as the states contained in this aggregates. In the simple example of Fig. 5(a), X initially contains three blocks, two of which contain a single element of type "macrostate" (aggregates S_0, S_2), the third contains three elements of type 'state' (states s_1, s_2 and s_3 of aggregate S_1). The notation used in the sequel for X is: $X = \{x_0 = \{S_0\}, x_1 = \{s_1, s_2, s_3\}, x_2 = \{S_2\}\}$.

$PreSplit$ (line 6) returns a refined set of X, such that each element A_i in this refined set satisfies the exact lumpability condition with respect to each element of X.

$$\forall s_1, s_2 \in A_i, \quad \sum_{s_k \in X_j, s_k \xrightarrow{\mu_{k1}} s_1} \mu_{k1} = \sum_{s_k \in X_j, s_k \xrightarrow{\mu_{k2}} s_2} \mu_{k2} \tag{3}$$

where $\mu_{k,i}$ represent the rate associated with the arc from s_k to s_i. Observe that this condition will be also the invariant of the iterative refinement phase.

The new refinement is obtained by splitting those sets of X that are reached by one or more instantiated arcs and/or such that one or more instantiated arcs depart from them. The splitting of such sets is performed considering the weights and source aggregates of the ingoing instantiated

[3]It will be clarified later how the initial partition is chosen and how the iterated refinement steps leading to each successive refinement work.

transitions, plus the global output rate of each state[4]. Finally (line 6), the new refinement is stored in A. In the simple example, A and X sets after the pre-splitting are: $A = \{a_0 = \{S_0\}, a_1 = \{s_1\}, a_2 = \{S_2\}, a_3 = \{s_2, s_3\}\}$

(2) The iterative refinement phase. The algorithm core consists of repeating a *refinement step* until X converges to A ($X = A$). The so-called refinement step is performed as follows:

In the partition X, an element D that has been refined in a previous step is selected (line 8), then the largest[5] element $B \in A$ s.t. $B \subseteq D$ is chosen (line 9). Finally, X is updated by replacing D with B and $D \setminus B$ (lines 10-11). In the example, X block x_1 is chosen and the A block a_3 is chosen in the role of B. All successors of B are computed and the following information are stored in Lel (line 12): the successor, the rate with which it is reached and the A-element index containing it. Then, the function $Partition\_wrt\_rate\_A$ performs a partitioning of Lel grouping the tuples with the same second and third element. In particular all the found "macrostate" elements are instantiated, so that the "macrostate" elements will be substituted by all the states that represent, and the generic arcs are instantiated using function f. In the example the "macrostate" S_2, reached by block a_3, is instantiated in s_4, s_5 and s_6, and $PartLel$ is $\{p_1\langle\{s_5, s_6\}, \beta, a_2\rangle\}$ (there is only one set of states, belonging to block a_2, reached by a_3 with rate β).
At this point, A must be refined according to the new partition represented by $PartLel$, as shown in Algorithm 3. In Algorithm 3, for each element $\langle S, rate, i\rangle \in PartLel$, we remove the elements of S from A_i. Line 6 replaces the old representation of A_i, while line 7 inserts S as a new set in A.

In the example the block a_2 is split in two blocks $a_2 = \{s_4\}$ and $a_4 = \{s_5, s_6\}$, so that list A is $\{a_0 = \{S_0\}, a_1 = \{s_1\}, a_2 = \{s_4\}, a_3 = \{s_2, s_3\}, a_4 = \{s_5, s_6\}\}$, while list X is $\{x_0 = \{S_0\}, x_1 = \{s_1\}, x_2 = \{s_4, s_5, s_6\}, x_3 = \{s_2, s_3\}\}$. The refinement is repeated choosing block a_4 in the role of B. This does not cause any new splitting: the final partition is the one of A above as illustrated in Fig. 5(b).

Observe that the algorithm may have to instance some states and arcs that will be aggregated again in the final CTMC, so that the peak of memory usage during execution exceeds the size of the final lumped CTMC, and may limit the applicability of the method.

Algorithm for strong lumpability check. It is easy to adapt the previous algorithm to check the strong lumpability condition instead of the exact lumpability one. In fact only a small part of the previous algorithm must be modified. First we need to modify slightly the pre-splitting phase. In line 6 the new refinement is derived by splitting those sets of X where one or more instantiated arcs depart from them. In the example of Fig. 5(a) the pre-split phase for strong lumpability provides the same result as the one already presented for exact lumpability.

In the refinement step only the following change is necessary: after selecting the block B and updating X, for every *Node* element in B we compute the elements reaching it, and the following information are stored in Lel (line 12): the predecessor, the rate with which predecessor reaches it and the A-element index containing predecessor.

[4]This is sufficient to assure the condition (3), because the subsets that are not reached by any instantiated arcs and/or such that no instantiated arcs depart from it do not need refinement, they already satisfy Eq. (3)

[5]In terms of number of contained elements.

$f_1 : f_1(s_0) = \{s_i, i = 1, 2, 3\}),\ \ f_2 : f_2(s_0) = \{s_i, i = 4, 5, 6\},\ \ , f_3 : f_3(s_i) = \{s_{i+3}, i = 1, 2, 3\}\ \ f_4 : f_4(s_i) = \{s_0, i = 4, 5, 6\}$

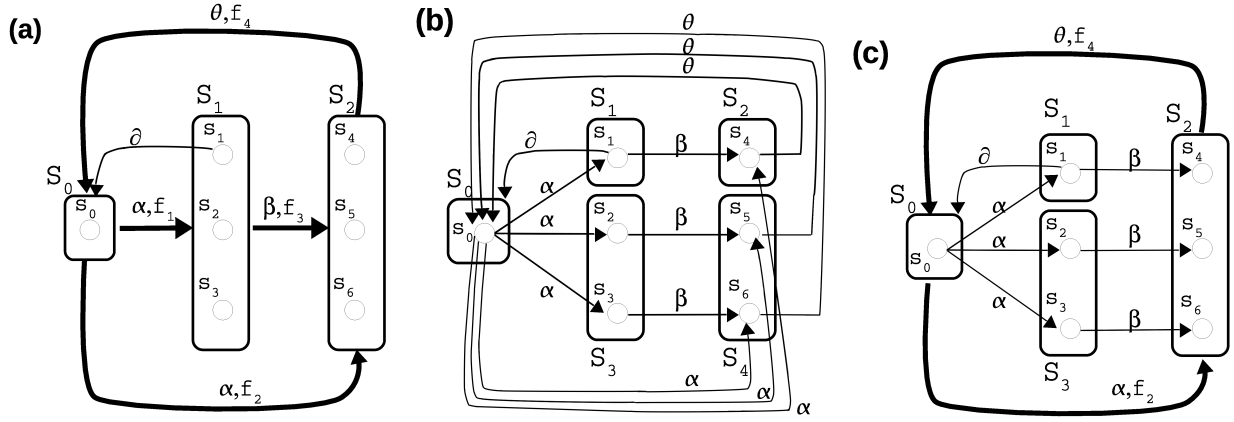


Figure 5: A simple example of almost symmetrical CTMC (a) and the result of exact (b) and strong (c) lumpability algorithm application.

After this, the refinement step works similarly to the exact lumpability version.

In the example a_3 is chosen again in the role of B but now the list $PartLel$ is: $PartLel = \{p1\langle\{s_0\}, \alpha, a_0\rangle\}$. This step requires the instantiation of aggregate S_0 and of the generic arc labeled α, f_1. Since S_0 contains only s_0 no split is performed and the algorithm ends.

Comparing the two final partitions (obtained using the two algorithms) we can observe that the strong lumpability condition for this example requires to instance less "macrostate" w.r.t the exact one. This is not always true, it depends on the characteristics of the model to be studied. For selecting the better one w.r.t. a particular model a first choice must be driven by the performance measures that we want to compute. If probabilities of individual markings (SMs) are needed, then the strong lumpability cannot be used since it only gives the probabilities of aggregates Instead if the performance measures can be expressed at the level of aggregates, then both approaches are suitable and a heuristic rule for defining the approach minimizing the number of instanced "macrostates" can be used.

Computation of the infinitesimal generator of the lumped CTMC. If the CTMC is strongly lumped, the infinitesimal generator is obtained by applying Equation 1.

In the case of exact lumpability, the splitting of an aggregate affects the computation of transition rates in the following way :

$$Q^{lp}(i,j) = \left(\frac{|S_j|}{|orig(S_j)|} \sum_{S_i \xrightarrow{\lambda, f} S_j} \lambda |f(s)|\ \right) + \left(\frac{|S_j|}{|S_i|} \sum_{s \in S_i : s \xrightarrow{\mu} s'} \mu\ \right) \tag{4}$$

where $s \in S_i$ and $s' \in S_j$, and $orig(S_j)$ denotes the aggregate to which states in S_j belonged in the initial almost symmetrical CTMC. This is needed because there might be generic arcs connecting a non split aggregate to an aggregate that has been split (there will be a replica of such generic arc for each refined aggregate substituting the original one).

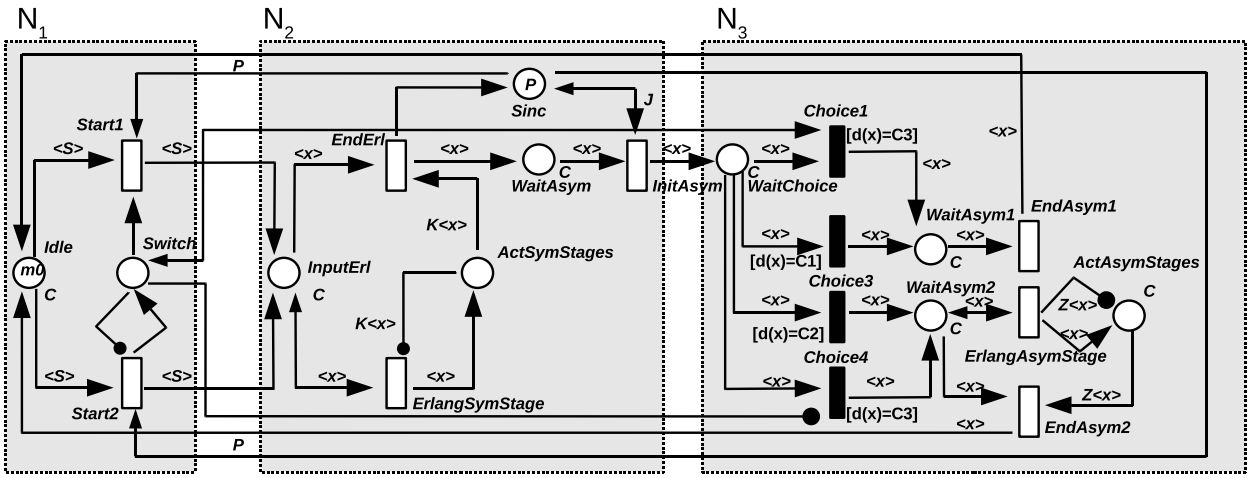


Figure 6: An example of SWN related to the Benchmark pattern.

In the examples of Figg. 5(b) and 5(c) the instantiated arcs created during the algorithm execution are shown instead of the arcs in the final lumped CTMC, to illustrate the algorithm operation and the memory peak problem. There will be only one arc between each pair of aggregates in the lumped CTMC, whose rate can be easily derived from the above formulae: e.g. in Fig. 5(c) the rate from S_0 to S_2 is 3α. Indeed, from Eq. 1 we obtain that $\sum_{S_0 \xrightarrow{\lambda,f} S_2} \lambda|f(s_0)|$) is 3α, since there is only one generic arc with rate α and $|f_2(s_0)| = 3$, and $\sum_{s' \in S_2 : s_0 \xrightarrow{\mu} s'} \mu$ is 0, since no instantiated arcs connecting any states of S_0 to S_2 exist. Instead, the rate from S_2 to S_0 is θ, since there is only one generic arc with rate θ and $|f_4(s_4)| = |f_5(s_5)| = |f_6(s_6)| = 1$ and no instantiated arcs.

In Fig. 5(b) the rate from S_0 to S_4 is 2α, indeed from Eq. 4 we obtain that $\sum_{S_0 \xrightarrow{\lambda,f} S_4} \lambda|f(s_0)|$=0 since no generic arcs connecting S_0 to S_4 exist, $\frac{|S_4|}{|S_0|} = 2$ and $\sum_{s \in S_0 : s \xrightarrow{\mu} s'} \mu = \alpha$ since only one instantiated arc exists between S_0 and S_4.

Finally, in the refinements in Fig. 4 we have both instantiated and generic arcs between the same aggregates. Then in Fig. 4(a) we observe that the rate from S_{i1} and S_j is $2\lambda + \mu$ because using Eq. 1 we obtain that $\sum_{S_{i1} \xrightarrow{\lambda,f} S_j} \lambda|f(s_i)|$) is 2λ since there is only one generic arc with rate λ and $|f(s_i)| = 2$, and $\sum_{s' \in S_j : s_i \xrightarrow{\mu} s'} \mu$ is μ since there is only one instantiated arc with rate μ. Instead, in Fig. 4(b), we observe that the rate from S_i and S_{j1} is $\frac{2\lambda}{3} + \frac{\mu}{3}$ because using 4 we obtain $\frac{|S_{j1}|}{|orig(S_{j1})|} = \frac{1}{3}$, $\sum_{S_i \xrightarrow{\lambda,f} S_{j1}} \lambda|f(s_i)|$ is 2λ since there is only one generic arc with rate λ and $|f(s_i)| = 2$, $\frac{|S_{j1}|}{|S_i|} =$ is $\frac{1}{3}$ and $\sum_{s \in S_i : s \xrightarrow{\mu} s'} \mu$ is μ, since there is only one instantiated arc with rate μ.

4. Instantiation of methods: the SWN formalism

4.1. Stochastic well-formed nets

Both methods handle sets of states which require a symbolic representation to efficiently manage them and a symbolic computation of the set of successors. Decision Diagrams (DD) could be used to this aim. However DD would not take into account that these sets are somewhat special

since they are orbits of subgroups. By contrast, some formalisms are tailored to take advantage of the symmetries during the modeling and the analysis stages.

Thus we have implemented our methods using the SWN formalism [5], a kind of high-level Petri nets that is the starting point of numerous efficient symmetry-based analysis methods already implemented in the GreatSPN tool [14]. Alternatively we could have instantiated our methods on symmetrical Stochastic Activity Networks [6] or on Stochastic Automata Networks [23].

Here we only describe the main features of SWNs illustrating them with the net of figure 6. In an SWN and more generally in a colored net, a *color domain* is associated with places and transitions. The colors of a place label the tokens contained in this place, whereas the colors of a transition define different ways of firing it. In order to specify these firings, a *color function* is attached to every arc which, given a color of the transition connected to the arc, determines the colored tokens that will be added to or removed from the corresponding place. Finally the initial marking is defined by a multi-set of colored tokens in each place.

The specification of the stochastic behavior is given first by associating priorities to transitions. Transitions with priority 0 are triggered by a negative exponential law whereas transition with non null priority are immediate (they fire in 0 time). Thus one associates a rate with transitions of priority 0 while one associates a weight with other transitions that is used in a random choice between enabled transitions with the same priority. Inhibitor arcs (with their usual meaning) are also used in order to obtain concise models.

SWN are a particular case of colored nets with a simple syntax. In SWN, a color domain is a Cartesian product of *color classes* which may be viewed as primitive domains. A class can be divided into *static subclasses*. The colors of a class have the same nature (e.g. processes) whereas the colors inside a static subclass have the same potential behavior (e.g. batch processes). In the net of figure 6, there is a single color class C and the color domains of places are either C or ε the neutral domain consisting of a single color (as in ordinary nets). For instance, the color domain of place $Switch$ is ε while the color domain of place $Idle$ is C. Class C models a set of tasks with three static subclasses $C = C_1 \uplus C_2 \uplus C_3$. C_1 is a set of interactive tasks, C_2 is a set of batch tasks and C_3 is a set of tasks that alternate between interactive and batch execution.

A color function is built by standard operations (linear combination, composition, etc.) on predefined basic functions. The most often used basic function is a projection which selects an item of a tuple and is denoted by a typed variable (e.g., x, y). Transitions can be guarded by expressions. An expression is a Boolean combination of predefined atomic predicates like $[x \neq y]$. In the net of Fig. 6, there are two functions. x is the identity over C meaning for instance that the color that instantiates transition $EndErl$ must be present in place $InputErl$ and it will be consumed by its firing. S is the constant color function that returns the bag of colors $\sum_{c \in C} c$ whatever the domain of the transition. For instance, in order to fire $Start1$, all task colors must be present in place $Idle$ and they will be moved in place $InputErl$. The guard $d(x) = C_2$ of transition $Choice3$ means that this transition may only be instantiated by a color of static subclass C_2. This net will be fully described in section 5.

4.2. Symbolic reachability graph buildings for SWN

The implicit symmetry of an SWN, obtained by its restrictive syntax, leads to a group G operating on color classes (and by extension on markings and firing instances). G is the intersection

of the isotropy subgroups of static subclasses. In other words, any permutation in G maps any static subclass onto itself. Given a marking m and a permutation g of G, the behavior of the net from the marking $g.m$ is the same as the behavior from m up to permutation g. The Symbolic Reachability Graph (SRG) construction lies on symbolic markings, namely a compact representation for a set of equivalent ordinary markings. A symbolic marking is a generic representation, where the actual color of tokens is forgotten and only their distributions among places are stored. Tokens with the same distribution and belonging to the same static subclass are grouped into a so-called *dynamic subclass*. Then, the SRG can be automatically built using a symbolic firing rule that directly applies to symbolic markings [5].

The critical factor for efficiency of the SRG method is the partition of a class into static subclasses. Finer is the partition, less effective is the reduction of the state space. Thus the implementation of both DS and TLS methods aims at keeping this partition as coarse as possible (locally).

In order to implement the DS method for a partially symmetrical CTMC, we specify this chain as the synchronized product of an SWN without static subclasses (so, G is the group of all permutations on color classes) representing the symmetrical MC, and a control automaton whose labels are sets of instances of transitions like $\{t(a,b), t(b,b), t(a,a), t(b,a)\}$ equivalently denoted $\bigvee_{x,y\in\{a,b\}} t(x,y)$. Thus the isotropy subgroup of a transition may be represented by a "local partition" in static subclasses (e.g. $\{\{a,b\},\{c,d\}\}$ for the label described above). The symbolic representation of a state of the lumped MC is then given by a local partition of color classes (corresponding to the isotropy subgroup of the set of associated states), a symbolic marking w.r.t. this partition, and a state of the automaton. The symbolic firing rule is close to the original one except that a refinement w.r.t. to the partitions of the synchronized transitions must be *a priori* performed, and a merging of static subclasses must be *a posteriori* performed in order to represent the isotropy subgroup of the new set of states. The graph which is built is called Dynamical SRG (DRSG), emphasizing that the partition in static subclasses depends on each node.

In order to implement the TLS method for an almost symmetrical CTMC, we specify this chain through an SWN where the transitions are split in symmetrical transitions, whose specification does not depend on static subclasses, and asymmetrical ones whose specification depends on them. An almost symmetrical CTMC is then generated from this specification. It corresponds to the Extended SRG (ESRG) [11] whose main feature is that a node has a two-level representation. At the higher level, a node is a symbolic marking w.r.t. the SWN without static subclasses: this symbolic marking is enough to check and fire symmetrical transitions. At the lower level, the symbolic marking is substituted by a set of symbolic markings taking into account the static subclasses partitions allowing to check and fire asymmetrical transitions. These two representations correspond to the same set of ordinary markings. The aim of the ESRG construction is to avoid developing the lower level representation for nodes as often as possible. This can be done when all ordinary makings of the node are known to be reachable and when none allows an asymmetrical firing (these conditions can be symbolically checked). Thus, the ESRG is the starting point of our adaptation of the Paige-Tarjan's algorithm where some avoided lower level representations are now developed (if needed) in order to meet the lumpability requirements.

The different approaches developed in this paper have been implemented in the GreatSPN tool (www.di.unito.it/∼greatspn) [14]. Actually, the kernel of the package, initially developed to

perform on global symmetries (the SRG approach), has been extended to handle dynamic and partial symmetries (the ESRG and DSRG approaches).

5. Analysis of efficiency: characterization of the appropriate models

5.1. Pattern characterization

As for every state-based reduction method, it is not easy to characterize the kind of models for which our methods bring a relevant reduction of the state space size. There are least two problems related with this characterization: (1) the "degree" of asymmetry of the system is not proportional to the number of asymmetrical transitions of the system but rather when they occur in the dynamics of the system. This was already experienced by the qualitative partially symmetrical methods; (2) the asymmetry may propagate due to the constraints associated with the lumping conditions.

In the light of these two problems and with the help of numerous models that we have tested since the development of these methods, we can suggest two application patterns for which our algorithms perform efficiently. The behavior of these systems can be schematized as an infinite loop where every cycle of the loop consists of:

- a synchronization stage where the consequences of the past asymmetrical behavior can be safely forgotten. This corresponds to reaching one or more "regeneration states" w.r.t. the history of asymmetrical behavior.
- a symmetrical and an asymmetrical stage that may overlap.

Here, we present two different patterns: the behavior of the first pattern is characterized as follows: a cycle of the loop consists of a synchronization stage followed first by a symmetrical stage and then by an asymmetrical one (not overlapping). The behaviors of the second pattern is characterized as follows: a cycle of the loop consists of a symmetrical stage followed first by a synchronization stage then by an asymmetrical one.

An example of net modeling the first pattern is shown in Fig. 6. This pattern models a system where a set of repetitive tasks are processed according to their types: interactive ones, batch ones and mixed ones. The first type represents the interactive tasks that prompt the user for such input, while the second represents the batch tasks that can be run to completion without human interaction. The last type represents tasks that alternate between interactive and batch executions.

The model can be divided in three parts: N_1, N_2 and N_3; where the submodel N_1 is the synchronization stage, N_2 the symmetrical stage and N_3 the asymmetrical one. In the submodel N_1 all tasks are simultaneously started (transition $start1$). Then, in the submodel N_2 every task perform some preprocessing whose time distribution is an Erlang-k where k is the number of stages. This is concisely modeled in the net with transitions ($ErlSymStage, EndErl$). When at least J tasks (with $1 \leq J \leq \#task$) have achieved their preprocessing the third part can start. In the third part (submodel N_3), every task acts depending on its type: the interactive activities are modeled by transitions $Choice3$ and $Choice4$ while the batch activities are modeled by transitions $Choice1$ and $Choice2$. As said before, the mixed tasks alternate between interactive and batch activities chosen at each start of cycle (when the synchronization stage is performed) with place $Switch$ controlling this alternation.

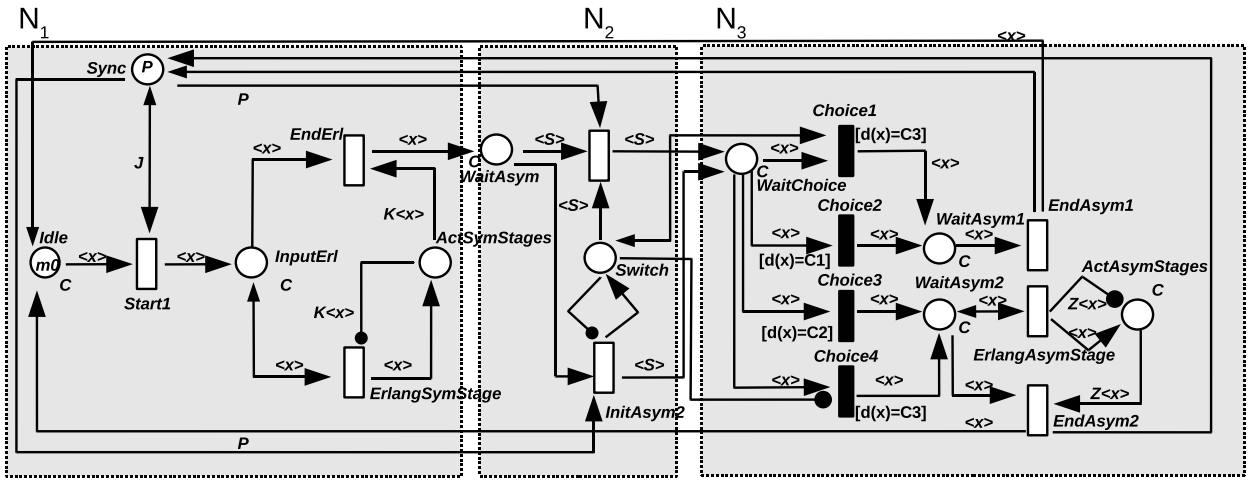


Figure 7: SWN model related to the Benchmark pattern.

The critical factor for the efficiency of the methods is the degree of overlapping of the symmetrical and asymmetrical behaviors. The more they are sequentialized the better the methods perform. The last point that can be emphasized is that the methods based on exact aggregation perform better when the symmetrical behavior starts first and vice versa for the method based on strong aggregation. This is certainly due to the way the constraints on lumping propagate (forward for exact lumpability, backward for strong lumpability).

In particular, in our example the critical parameter is J since it rules the overlapping of the symmetric and asymmetric behaviors. If $J = \#task$ then there is a complete sequentialization whereas if $J = 1$ there is a complete overlapping.

Let us examine the experiment results on the net of Fig. 6 summarized in Table 1. The first column shows the values of the experiment parameters: the number of stages of the Erlang (K), the level of overlapping of the symmetrical and asymmetrical behaviors (J), the number of tasks for each type. The Columns labeled "St." represent the number of constructed states for each structure [6]. The columns labeled "Peak" contain the total number of intermediate states stored to obtain the final structure (only for ESRG and RESRGs). Finally the seventh, tenth and twelfth columns show the reduction factor obtained using these three methods.

We have to highlight that a good level of reduction is obtained by both the approaches when there is a complete sequentialization between the symmetric and asymmetric behaviors. For instance, the experiment with parameters $10-8-3-3-2$ leads to a reduction factor of approximately 120 whatever method.

In Fig. 7 an example of net modeling the second pattern is shown. This model is obtained by the previous one (Fig. 6) discarding the initial synchronization and inserting it just before the sym-

[6] for the ESRG are also shown the number of Extended Symbolic Marking (ESM, first level representation) plus the Eventualities (Ev, second level representation)

	SRG	ESRG		RESRG (Exact)			RESRG (Strong)			DSRG	
K, J, |C1|, |C2|, |C3|	St.	St.(Esm. + Ev.)	Peak	St.	Peak	$\frac{SRG}{RESRG}$	St.	Peak	$\frac{SRG}{RESRG}$	St.	$\frac{SRG}{DSRG}$
6-6-2-2-2	97,495	4,255+456	4,264	7,138	7,615	13.65	4,460	7,588	21.85	8,179	11.92
6-4-2-2-2	247,498	16,393+40,044	715,321	93,118	157,304	2.65	19,913	154,602	12.42	150,125	1.64
6-2-2-2-2	724,758	39,157+67,322	499,014	469,442	606,258	1.54	49,251	606,458	14.71	427,255	1.69
8-6-2-2-2	336,933	10,833+823	10,323	13,716	14,193	24.56	11,038	14,166	30.52	14,757	22.83
8-4-2-2-2	571,293	29,139+30,934	215,217	144,210	246,497	3.96	34,349	248,895	16.63	235,248	2.45
6-8-3-3-2	1,054,508	15,224+129,002	34,724	29,837	30,574	35.20	25,933	30,529	40.66	30,463	34.61
6-6-3-3-2	1,969,954	58,547+13,770	897,166	666,669	944,634	2.95	72,497	939,113	27.17	933,475	2.11
10-6-2-2-2	953,287	25,575+799	23,004	28,458	28,931	33.49	25,780	28,908	36.97	29,499	32.31
12-6-2-2-2	2,319,433	55,087+1,345	46,773	57,970	58,443	40.01	55,292	58,420	41.94	59,011	39.30
14-6-2-2-2	5,035,095	109,351+2,840	87,829	112,234	112,707	44.86	109,556	112,684	45.95	113,257	44.45
10-8-3-3-2	20,687,084	153,518+7,654	406,776	168,131	168,868	123.041	154,227	168,823	134.13	168,757	122.58
12-8-3-3-2	-	out of memory	-	-	-	-	-	-	-	424,573	-

Table 1: Experiment results on the net of Fig. 6

	SRG	ESRG		RESRG (Exact)			RESRG (Strong)			DSRG	
K, J, |C1|, |C2|, |C3|	St.	St.(Esm. + Ev.)	Peak	St.	Peak	$\frac{SRG}{RESRG}$	St.	Peak	$\frac{SRG}{RESRG}$	St.	$\frac{SRG}{DSRG}$
6-6-2-2-2	182,690	6,466+679	5,987	6,362	6,446	28.71	6,195	6,446	29.48	6,466	28.25
6-4-2-2-2	524,050	26,355+10,345	341,798	492,786	524,050	1.06	26,456	347,806	19.06	121,791	4.30
6-2-2-2-2	569,926	32,915+23,456	387,674	561,573	569,926	1.04	33,016	393,682	17.26	133,619	4.26
8-6-2-2-2	575,432	16,197+1,123	14,882	16,372	16,456	35.08	16,205	16,456	35.50	16,476	34.92
8-4-2-2-2	1,417,537	60,213+45,457	842,543	1,347,762	1,417,537	1.05	60,338	858,561	23.49	60,492	23.43
6-8-3-3-2	2,451,382	26,128+3,400	64,938	26,872	26,872	91.22	26,156	26,872	93.72	26,800	91.46
6-6-3-3-2	-	out of memory	-	-	-	-	-	-	-	138,312	-
10-6-2-2-2	1,507,582	37,309+10,342	32,483	37,484	37,568	40.21	37,317	37,568	40.39	37,588	40.10
12-6-2-2-2	3,456,440	77,701+15,334	64,001	77,876	77,960	44.38	77,709	77,960	44.47	77,980	44.32
14-6-2-2-2	7,163,594	149,407+18,342	116,610	149,582	149,666	47.89	149,415	149,666	47.94	149,686	47.85
10-8-3-3-2	-	out of memory	-	-	-	-	-	-	-	223,003	-
12-8-3-3-2	-	out of memory	-	-	-	-	-	-	-	561,136	-

Table 2: Experiment results on the net of Fig. 7

metrical part. Hence, the submodel N_1 represents the symmetrical stage, N_2 the synchronization stage and N_3 the asymmetrical one.

The experiment results on this net are summarized in Table 2, where the columns have the same meaning of the columns in the Table 1. The same considerations discussed previously hold true, therefore the efficiency of the two approaches seems not to be influenced by the stage order inside the round of the loop, but by the degree of overlapping among the symmetrical and asymmetrical behaviors.

5.2. Expected efficiency of the behavior of the methods

We can compare the proposed methods w.r.t. different criteria. The size of the lumped chain is generally smaller with the TLS method as it is based on the minimization of a MC w.r.t. lumpability. However the TLS method requires to explicitly develop "asymmetrical" set of states during the refinement process thus facing the problem of a peak in memory usage, contrary to the DS method. Moreover considering instantiation of the TLS method w.r.t to the SWN formalism, a further peak in memory usage raises during the ESRG generation; in fact the ESRG algorithm maintains explicitly the lower-level representation of reached markings until the set can be compacted into a symmetrical higher-level representation.

The memory peak in the refinement process can be reduced by discarding all lower-level nodes in a block and rebuilding them when needed, while the ESRG memory peak cannot be mitigated, so that it may prevent a solution to be reached (e.g. the case $12-8-3-3-2$ in Table 1, the ESRG cannot be computed due to its memory occupation peak).

On the one hand, when a model is efficiently handled by the methods, the final sizes are of the same magnitude order. On the other hand, when the asymmetry of the model propagates throughout the state space, it may yield a combinatorial explosion and the size of the lumped chain of the DS method may become bigger than the original one. This cannot happen by construction with the TLS method. Notice that in [24] an extension of the DS method has been proposed in order to cope with this problem. Finally, the DS method is parametrized in the following sense: as lumping is based on labels the modeler can freely change the numerical values associated with labels without need to recompute the graph associated with the lumped chain; only the numerical values have to be updated. In order for the TLS method to support such a parametrization the refinement algorithm must be based on transition labels instead of rate values.

In the next section new experiments are presented, referring to models of actual systems rather than abstract patterns, and the above considerations verified in a more applicative setting.

6. Case studies

6.1. A readers writers example

The readers writers example in Fig. 8 models a database which is accessed by users for read or write operations. Readings may be simultaneous while writings are mutually exclusive w.r.t both the readings and the writings. Thus as soon as a writing is queued, it waits for all the readings to end before performing its transaction without concurrent accesses.

Readings are operations with less variability than writings. Thus we have chosen to represent their distribution by an Erlang distribution while the writing have two exponential distributions depending on the class of users: ordinary ones that perform unitary updates while administrators

perform a batch of updates (given for instance by an auxiliary file).

Let us map this model on the patterns presented in the previous section: the synchronization points occur after every writing. Then there is a (possibly empty) stage of readings which corresponds to the symmetrical part of the behavior (submodel N_1). The asymmetrical part of the behavior (submodel N_2) starts at the beginning of a writing since only in that case the kind of user matters. Observe that there is no overlapping between the two stages since the readings end before the writing starts. So the results are very good as witnessed by Table 3. The experiment parameters are (in order) the number of stages of the Erlang distribution, the number of administrators and the number of ordinary users (see the first column).

Moreover the columns labeled "St." represent the number of constructed states for each structure. The columns labeled "Peak" contain the total number of intermediate states stored to obtain the final structure (only for ESRG and RESRGs). Finally the seventh, tenth and twelfth columns show the reduction factor obtained using these three methods.

Observe that the parameter that has more impact on the reduction factor is the number of system users (ordinary and administrator users), so that the reduction factor increases w.r.t to this parameter. For instance the experiment with 13 users ($|M| + |N| = 3 + 10$) leads to a reduction factor of approximately 30 whatever the method.

6.2. A client-server example

The client-server example, in Fig. 9, is composed of a finite number of terminals and a Remote Terminal Server (RTS). Via a terminal, a client tries to open a connection with the RTS. This connection is accepted if the maximum load of the RTS has not been reached yet and then it is authenticated. Once authenticated, a client asks for a service that can be *non-critical* (submodel N_3) or *critical* (submodel N_4, N_5 and N_6). Non-critical services can be handled simultaneously, while a critical service must be performed in mutual exclusion with any other service. The system ensures a weak priority for non-critical services based on a *wave* mechanism. The wave consists of the clients currently accepted by the RTS (submodel N_2). Once a client chooses a critical service accepted by the RTS, no more clients can join the wave. Critical services are performed only when there are no more clients in the authentication stage or in a non-critical service execution. When the last critical service of the wave completes, a new wave can start. For efficiency reasons, during a wave the RTS accepts a limited number of different concurrent user classes in the critical services. A critical service request related to a user class not yet in competition is rejected if the maximum number of concurrent user classes has been reached.

A critical service is divided into two sequential stages: a preprocessing step that can be performed concurrently and a main step that is performed in mutual execution. A priority rule is applied and the requests access the critical section following the order of the user classes. Observe that in this case the first critical service that has achieved its preprocessing step must wait if it does not belong to the highest priority user class in competition.

This example is a refinement of the previous one when we consider the critical services as write processes and the non-critical as read processes. First the synchronization step occurs at the end of a wave which includes multiple critical services. Furthermore the overlapping of the asymmetrical part of the behavior and the symmetrical one is more important since the preprocessing step of critical service is symmetrical whereas the entrance in the critical section is asymmetrically

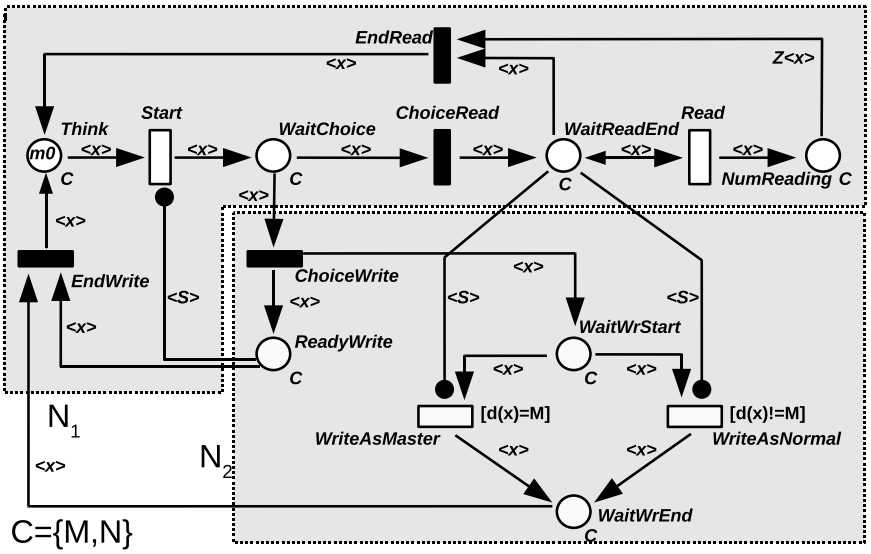


Figure 8: SWN model implementing the Reader-Writer pattern.

	SRG	ESRG		RESRG (Exact)			RESRG (Strong)			DSRG	
Z,|M|,|N|	St.	St.(Esm. + Ev.)	Peak	St.	Peak	$\frac{SRG}{RESRG}$	St.	Peak	$\frac{SRG}{RESRG}$	St.	$\frac{SRG}{DSRG}$
6,1,3	1,535	491+3	141	492	492	3.12	491	491	3.12	491	3.12
6,1,6	21,338	4,951+3	1,263	4,952	4,952	4.30	4,951	4,951	4.30	4,951	4.30
6,1,8	75,968	15,731+3	3,625	15,732	15,732	4.82	15,731	15,731	4.82	5,732	4.82
6,1,10	216,218	41,407+3	8,682	41,407	41,407	5.22	41,406	41,406	5.22	41,406	5.22
6,2,6	97,358	9,076+3	5,387	9,077	9,077	10.72	9,076	9,078	10.72	9,076	10.72
6,2,8	344,192	26,027+3	15,716	26,028	26,028	13.22	26,027	26,027	13.22	26,028	10.72
6,2,10	974,976	63,701+3	37,896	63,702	63,702	15.30	63,701	63,701	15.30	63,701	15.30
6,3,6	321,638	15,731+4	16,641	15,732	15,732	20.44	15,731	15,731	20.44	15,731	20.44
6,3,8	1,131,671	41,406+4	49,369	41,407	41,407	22.92	41,406	41,406	22.92	41,406	22.92
6,3,10	3,195,194	95,201+4	120,085	95,202	95,202	33.56	95,201	95,201	33.56	95,202	33.56
8,1,6	77,816	16732+3	3,667	16,733	16,733	4.65	16,732	16,733	4.65	16,732	4.65
10,1,6	228,230	46,476+3	8,985	46,477	46,477	4.91	46,476	46,476	4.91	46,476	4.91
12,1,6	574,394	112,269+3	19,427	112,270	112,270	5.11	112,269	112,269	5.11	112,269	5.11

Table 3: Results for the Reader-Writer example in Fig. 8.

managed. However the propagation of asymmetry depends on two factors: the maximal load of the RTS and the maximum number of simultaneous user classes.

Let us examine in more details Table 4. The first column shows the experiment parameters: $Local$, GC, LC, $Prio$.

- $Local$: represents the number of terminals, affects only the "frontal" behavior of the system (outside the server). The increase of the number of terminals does not affect the internal activities of the server
- $GC(\leq Local)$: represents the maximum number of users allowed to simultaneously access the server. This parameter has crucial impact on the global behavior of the server. Actually, its value affects the symmetric part as well as the asymmetric one.
- $Prio$: represents the cardinality of the color class C. Since there is a bijection between this cardinality and the number of different priority classes, we observe a strong dependency between the value of this parameter and the efficiency of the different approaches.
- $LC(\leq Prio)$: the maximum number of user classes allowed to access, simultaneously, the critical part. The value of this parameter will affect, essentially, the behavior of the asymmetric part (with some side effect on the symmetric part).

The other columns have the same meaning of the columns in the previous table.

We notice here the significant reduction achieved by the DSRG and ESRG w.r.t. the SRG. For instance, in case 8,5,3,5 the ESRG reduction factor (both strong and exact) is 45.53, while the DSRG reduction is 27.95. However, we remark that in this model the memory peaks of the two approaches based on the ESRG have a high impact. For instance, for 8,5,3,8 the ESRG cannot be computed due to its computation peak. Hence the final aggregation obtained by the two approaches based on the ESRG is better, but that the number of real ESMs and eventualities stored during the computation is greater than that of the DSRG.

6.3. A workflow example

A workflow is a set of tasks organized through a model that describes the triggering conditions for every task. It is usually described by operators like sequential flow, parallel flow, choice flow, etc. that are easily modeled by an ordinary Petri net. In addition, with every task is associated a set of agents which are allowed to perform it. A job is an instance of a workflow and we consider simultaneous executions of jobs with the same workflow.

Our experiment is based on the fixed control flow shown in the SWN model in Fig. 10. The asymmetry is due to the set of agents which are divided in groups $\{G_i\}_{i\in\{1,2,3\}}$, according to their authorization levels: an agent $g \in G_i$ has less authorizations than an agent $g' \in G_j$ s.t. $j > i$. We specify for every task T an execution threshold $i(T)$: an agent in G_i is allowed to perform task T if $i \geq i(T)$.

The execution cost of a task depends both on the execution time and on its execution threshold. Thus this model enforces a policy that aims at minimizing the execution time of special tasks with a high execution threshold. So with every execution threshold we associate a maximum number

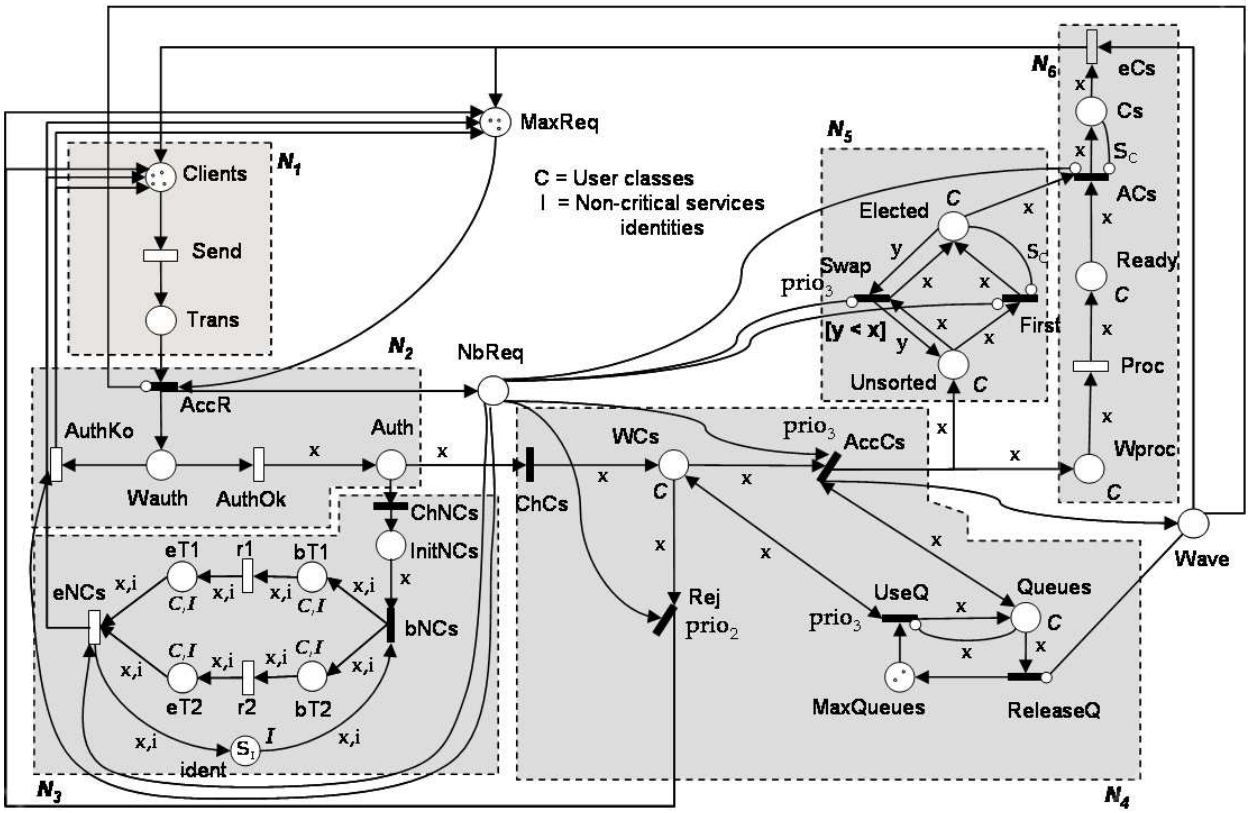


Figure 9: SWN model representing a client-server with critical section example.

	SRG	ESRG		RESRG (Exact)			RESRG (Strong)			DSRG	
Local, GC, LC, Prio	St.	St. (Esm. + Ev.)	Peak	St.	Peak	$\frac{SRG}{RESRG}$	St.	Peak	$\frac{SRG}{RESRG}$	St.	$\frac{SRG}{DSRG}$
3,3,2,5	19,108	981 + 470	4,050	998	1,727	19.14	998	1,727	19.14	1,217	15.70
3,3,2,8	72,772	981 + 1,316	17,028	998	3,185	72.91	998	3,185	72.91	1,436	50.67
3,3,3,3	4,778	1,028+179	846	1,064	1,128	4.49	1,060	1,228	4.49	1,142	4.18
3,3,3,5	19,918	1,028+850	4,080	1,064	2,326	18.71	1,060	2,326	18.71	1,613	12.34
3,3,3,8	77,308	1,028+ 3,444	17,196	1,064	6,498	72.93	1,060	6,498	72.63	2,765	27.95
8,5,3,3	496,618	90,429 + 4,522	59,187	92,600	96,088	5.36	92,224	96,088	5.36	94,593	5.25
8,5,3,5	4,788,499	108,205 + 28,040	691,390	110,376	159,104	45.53	110,000	159,104	45.53	134,553	35.58
8,5,3,8	-	out of memory	-	-	-	-	-	-	-	193,401	-

Table 4: Results for the clients-server example in Fig. 9

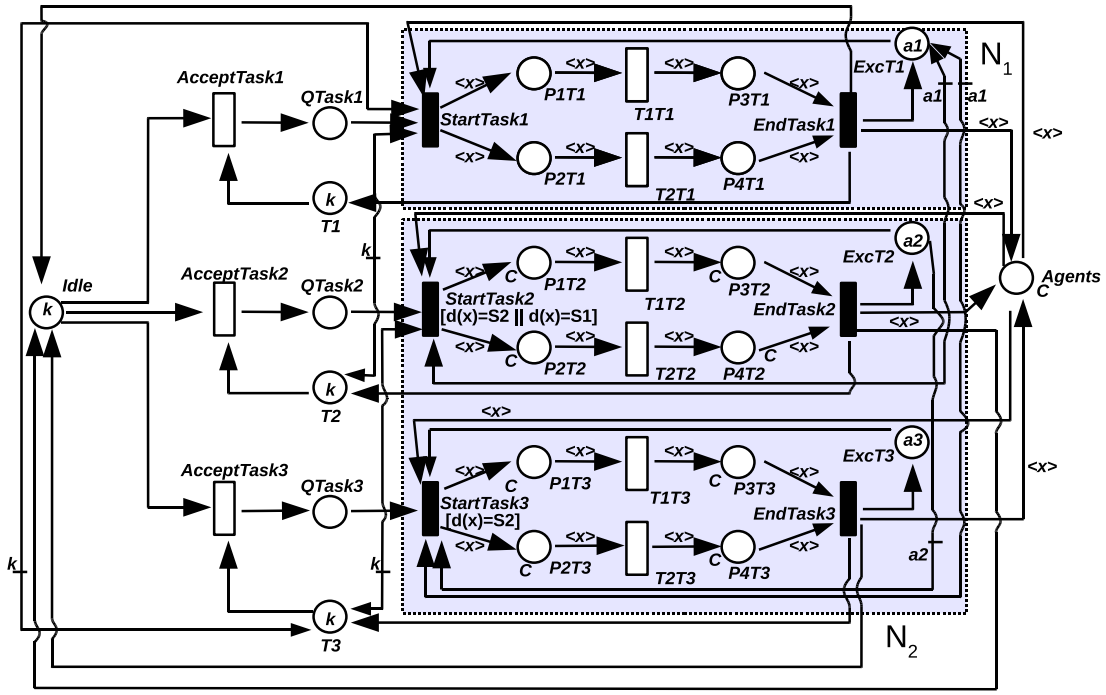


Figure 10: SWN of the workflow example.

of simultaneous executions. Then, when a task is triggered, it is queued and later on it is executed. A task is executed if the following conditions are fulfilled:

- there is no running or waiting special task with higher execution threshold;
- there are only running tasks with the same execution threshold;
- the maximum number of simultaneous executions corresponding to its execution threshold is not reached.

Most of the tasks can be executed by every agent, we call it *normal tasks*; the other ones are called *special tasks*. The symmetrical behavior corresponds to a wave of executing normal tasks (submodel N_1) whereas the asymmetrical behavior corresponds to a wave of special task executions with the same execution threshold (submodel N_2). The synchronization steps occur after every wave execution. Observe here that between two synchronization steps there is either a symmetrical behavior or an asymmetrical one but not both.

Let us examine in more details Table 5, that shows some experiments performed on this model for different values of its parameters: the number of total tasks (K), the maximum number of simultaneous executions for each threshold (a_i), the number of normal agents ($|G_1|$) and the number of special agents ($|G_2| + |G_3|$). We observe that both these methods reach the same reduction factor; moreover the parameters that have more impact on the reduction factor are the number of total tasks and the maximum number of simultaneous executions for normal tasks. The reduction factor is increasing w.r.t to these two parameters.

	SRG	ESRG		RESRG (Exact)			RESRG (Strong)			DSRG	
K, a1, a2, a3, $\|G_1\|$, $\|G_2\|+\|G_3\|$	St.	St.(Esm. + Ev.)	Peak	St.	Peak	$\frac{SRG}{RESRG}$	St.	Peak	$\frac{SRG}{RESRG}$	St.	$\frac{SRG}{DSRG}$
5-4-2-1-2-2	3,550	1,003+274	682	1,003	1,244	3.53	1,003	1,244	3.53	1,003	3.53
10-4-2-1-2-2	40,060	9,643+2,199	5,447	9,643	11,444	4.15	9,643	11,444	4.15	9,643	4.15
15-4-2-1-2-2	151,045	34,933+6,614	17,417	34,933	40,869	4.32	34,933	40,869	4.32	34,933	4.32
5-4-3-2-2-4	7,994	1,429+1044	1,878	2,437	2,440	3.28	1,429	2,440	5.59	1,429	5.59
10-4-3-2-2-4	108,789	13,889+10,504	17,708	24,147	24,175	4.50	13,889	24,175	7.82	13,899	7.82
15-4-3-2-2-4	431,134	50,399+38,289	63,413	87,932	88,010	4.90	50,399	88,010	8.55	50,399	8.55
15-4-2-1-3-2	200,467	34,933+6,614	23,483	34,933	40,869	5.73	34,933	40,869	5.73	34,933	5.73
15-6-2-1-4-2	383,108	55,497+6,814	40,776	55,497	61,433	6.90	55,497	61,433	6.90	55,497	6.90
15-8-2-1-6-2	655,095	74,951+7,023	62,540	74,951	80,887	8.74	74,951	80,887	8.74	80,887	8.74
15-10-2-1-8-2	885,630	89,075+7,467	70,031	89,075	95,011	9.94	89,075	95,011	9.94	95,011	9.94
15-12-2-1-10-2	1,026,425	96,401+9,453	71,517	96,401	102,337	10.64	96,401	102,337	10.64	102,337	10.64
15-15-2-1-13-2	1,086,911	98,556+10,345	71,573	98,556	104,492	11.02	98,556	104,492	11.02	98,556	11.02

Table 5: Results for the workflow example in Fig 10

6.4. A cluster computing example

This pattern corresponds to a finite set of machines grouped in clusters. Each cluster has *a master* machine and a set of *slave* machines. Every machine can fail while being idle or working. While idle, the failing of a slave machine means its removing from the cluster, for (local) updating/maintenance reasons. It is put back in its environment as soon as it is reconfigured. Instead, if it fails while working because of a hardware/software problem then its last stable state is saved. Afterwards it is repaired and finally restarted.

The failing of a master has a different consequence: the whole cluster is no longer reachable. Actually, even if the slave machines of the cluster are not in a fail state, the absence of a master makes the cluster in an unstable state. In such a case, the cluster becomes unavailable until the recover of the master machine.

The system presents a symmetrical behavior until the first failure. However after this failure we cannot identify synchronization steps. Thus the achieved reduction is poor as detailed in Table 6. The SWN model implementing the cluster computing pattern is shown in Fig. 11. It is divided in four submodels: N_1, N_2, N_3 and N_4. N_1 models the jobs submission and the cluster/machine assignment, N_2 the machines and clusters states, N_3 the correct job execution (without failure), and N_4 the job failure due to the failure of the machine or the cluster where it is running.

Several experiments have been performed on this SWN model for different values of the system parameters: the number of jobs ($|Job|$), the number of clusters ($|Cl|$) and the number of machines per cluster ($|M|$), but the best result obtained for the reduction factor is less than 3. It is worth noting that the number of ESM and eventualities stored during the generation of the ESRG and during the refinement steps is close to the $|SRG|$; moreover the DSRG size in the worst case is higher than the SRG one (e.g. case 16-2-8-2).

Remarks. The reader may refer to [25] for a detailed description of all the models presented in this section. Let us add few comments on the computation times of the experiments performed on these case studies and on the benchmark models. In general, it is not easy to forecast the required memory and time resources required by each method for a given SWN model just analysing its structure From the experimental results we observed that the DSRG computation time is lower than the ESRG one for the reader-writer model and those experiments on the benchmark models

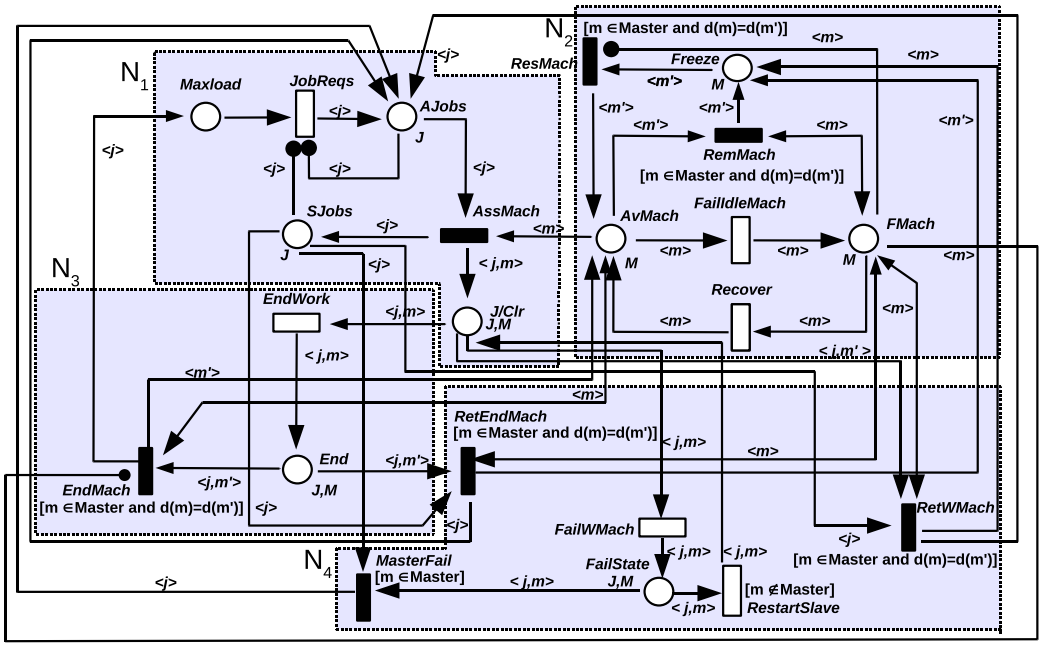


Figure 11: SWN representing an example of cluster computing.

when there is a complete sequentialization between the symmetric and the asymmetric behaviors. For instance, for the case 6-1-8 of reader-writer model the DSRG computation time is 633s, while the ESRG(exact) one is 871s.

Instead, the DSRG computation time is higher than the ESRG one for the workflow model, the client-server model and those experiments on benchmark models when there is no complete sequentialization between the symmetric and the asymmetric behaviors leading to a number of states in the DSRG significantly above the number of ESMs. For the case 6-6-3-3-2 of the first benchmark model the DSRG computation time is $\sim$57h, while the ESRG(exact) one is $\sim$38h.

Finally, in the cluster computing example the DSRG computation is considerably higher than ESRG one. For instance, for the case 5-2-5-3 the DSRG computation time is 148s, while the ESRG one is 32s.

7. Application of the methods to other models

7.1. DS for Stochastic Automata Networks

Stochastic Automata Networks [23] describe a system as a set of subsystems that interact. Each subsystem is modeled by an automaton with stochastic features such that, given an appropriate semantics, the whole system is a CTMC. More precisely, every transition is labeled by a *local* or a *synchronized* event. A synchronized event occurs in several automata, one of them being its *triggering* automaton; a local event only occurs in its triggering automaton. In addition, every transition is labeled by a rate in the corresponding triggering automaton. At last, the rate may be a function of the *global state*. In a global state (*i.e.* one state per automaton) an event is enabled when there is a possible transition in the states of the associated automata. Its rate is then obtained by applying the corresponding function to the current state. The interest of this model

	SRG	ESRG		RESRG (Exact)			RESRG (Strong)			DSRG	
|Job|,|Cl|,|M|	St.	St. (Esm. + Ev.)	Peak	St.	Peak	$\frac{SRG}{RESRG}$	St.	Peak	$\frac{SRG}{RESRG}$	St.	$\frac{SRG}{DSRG}$
2,2,2,2	1,803	220+1,601	1,634	897	1,803	2.10	625	1,795	2.84	2,008	0.9
5,2,2,3	3,776	408+3,318	3,352	1,790	3,776	2.10	1,254	3,703	3.01	3,646	1.03
5,2,2,4	6,295	641+5,468	5,330	2,790	6,295	2.25	1,861	6,080	3.38	5,771	1.09
5,2,2,5	9,002	900 +7,642	7,202	3,712	9,002	2.42	2,474	8,507	3.63	7,565	1.18
5,2,5,2	15,369	910+14,457	15,198	7,625	15,369	2.01	6,752	15,350	2.27	15,343	1
5,2,5,3	35,246	1,779+33,415	34,788	17,293	35,246	2.03	13,117	35,171	2.68	35,037	1
5,2,5,4	66,413	2,971+63,226	65,162	32,217	66,413	2.06	20,602	66,168	3.22	65,575	1.01
5,2,5,5	109,118	4,483+103,947	105,744	52,039	109,118	2.09	35,032	108,486	3.11	106,374	1.02
16,1,8,2	734	459+327	575	733	741	1	707	730	1.03	731	1
16,2,4,2	8,797	627+8,168	8,626	8,615	8,797	1.02	4,106	8,778	2.14	9,105	0.96
16,1,16,2	2,418	1,547+971	2,124	2,409	2,433	1.00	2,381	2,414	1.01	2,415	1.00
16,2,8,2	52,713	2,077+50,634	52,542	51,919	52,713	1.00	22,406	52,694	2.35	53,010	0.99

Table 6: Results for the cluster computing example in Fig. 11

lies in the expression of its infinitesimal generator which is a tensorial expression of matrices whose dimension are the the size of the local state spaces thus leading to a drastic reduction of the required memory. In [4], the model is specialized: Stochastic Automata Networks with replicas are defined by a partition of automata. Inside a set of the partition called a *replica*, the automata have the same behavior. Furthermore synchronization and functions of rates must be symmetric. In that case, a local lumping is possible and one still has a tensorial expression where the matrices are now related to the aggregation inside the replicas.

In order to allow partial symmetry and still obtain a tensorial expression, asymmetries should only occur in local events. The asymmetries could be defined by a set of control automata, one per replica that never disable synchronizing event. With this assumption, we could apply a specialization of algorithm 1.

Observe that this algorithm operates at a symbolical level and thus does not require any rate computation in order to build $\mathcal{G}_{\mathcal{A}}$. The specialization is easier than the one for SWNs and could be seen as a particular case with a single class. On the contrary, once the aggregated states are built, the computation of the rates of the local matrices is more involved but can be obtained following the method of [4].

7.2. TLS for Stochastic Activity Networks

The Stochastic Activity Network formalism [6] is a Petri Net like language with features that make it easier to model quite complex systems. The feature that is more relevant in the context of this paper is the way models can be composed from submodels and how this can be exploited to build a more compact reachability graph and corresponding CTMC [6, 7]. A Stochastic Activity Network model is defined hierarchically by instantiating subnets, and joining subnets on shared places. When a subnet is instantiated it can be replicated several times: since the replicas are identical, their state can be described in a compact way, namely, instead of keeping track of the state of each single (and uniquely identified) subnet in the set, only the number of subnets in each state is recorded, disregarding the information on which specific subnet is in a given state. This is a type of symmetry that could be easily captured by the SRG algorithm on an SWN-like version of the Stochastic Activity Network model (by using colored places and transitions in the subnet to be replicated). Although the subnets replication mechanism offered by Stochastic Activity Network

is less powerful than using colors, from the point of view of the modeler this mechanism can be more intuitive and easier to use, and it should be used instead of colors whenever appropriate. Also the more abstract state representation is simpler than the symbolic marking one devised for the SWNs. A little extension to the Stochastic Activity Network formalism may allow to take into account situations leading to partial symmetries and in this case the TLS method could be applied.

The extension could be as follows: when specifying a subnet it could be associated with a set of "versions", e.g. $Versions = \{A, B, C\}$; the transitions in the subnet are annotated with subsets of $Versions$ (the default being the whole set $Versions$). The intended meaning is that we have slightly different versions of the same subnet (e.g. the various versions may have identical structure but a few transitions with different rate), and that the transitions enabling is conditioned on the version a given subnet belongs to. The replicate operator in the extended formalism must provide the number of instances of the subnet for each element in $Versions$. This extension introduces something similar to static subclasses in the SWN color classes. In the current Stochastic Activity Network formalism such a situation would require to define separate submodels and to use several replication operators, one for each "subnet version".

On such extended Stochastic Activity Network formalism the TLS method could be applied: indeed the more abstract state representation is the one that considers all subnets as completely symmetric. As long as only symmetric transitions are enabled (i.e. the ones annotated with the whole set $Versions$) then the abstract state representation can be kept, and transition can be fired from such representation leading to another abstract representation; only when at least one of the asymmetric transitions (i.e. one transition annotated with a proper subset of $Versions$) is enabled, then a refinement is needed, and the different refined markings separating the various subnet versions are generated: the asymmetric transitions are fired from the more refined markings, and lead to refined markings (associated with a more abstract representation). The final structure derived by applying this procedure is similar to the ESRG for SWNs, and it must be checked against one lumpability condition and possibly refined to obtain a proper lumped Markov chain.

8. Conclusion and future work

In this paper the DS and TLS methods have been presented: their goal is to generate a lumped CTMC from a partially symmetrical and almost symmetrical CTMC specification respectively. They can be efficiently applied only if the MCs on which they operate can be handled symbolically, exploiting the a *priori* known presence of symmetries: this happens when they are derived from a higher level model, such as an SWN, where the presence of similarly behaving components is made explicit. Implementation issues have also been discussed referring to SWNs. Six case studies are presented to show the methods effectiveness and their applicative interest. Moreover we have presented a characterization of the type of models that can fully exploit the potential of the presented methods, based on their structural properties.

A possible line of development is to complement these methods with the possibility of computing bounds on the performance indices by transforming the partially symmetrical MC into a symmetrical one, and using stochastic ordering arguments. Finally the presentation of the methods in a general setting could be a good starting point to extend their application to other high level formalisms able to highlight the presence of similarly behaving components: two examples of application have been suggested in the paper.

[1] P. Huber, A. M. Jensen, L. O. Jepsen, K. Jensen, Towards Reachability Trees for High Level Petri Nets, in: Proc. of EWATPN, Aarhus, Denmark, 1984.
[2] C. Norris, D. L. Dill, Better verification through symmetry, FMSD 9 (1/2) (1996) 41–75.
[3] E. Emerson, A. Prasad Sistla, Symmetry and Model Checking, FMSD'96 9 (1996) 307–309.
[4] A. Benoit, L. Brenner, P. Fernandes, B. Plateau, Aggregation of Stochastic Automata Networks with replicas, Linear Algebra and its Applications 386 (2004) 111–136.
[5] G. Chiola, C. Dutheillet, G. Franceschinis, S. Haddad, Stochastic well-formed coloured nets for symmetric modelling applications, IEEE Transactions on Computers 42 (11) (1993) 1343–1360.
[6] W. Sanders, J. J.F. Meyer, Reduced Base Model Construction Methods for Stochastic Activity Networks, IEEE Journal on Selected Areas in Communications 9 (1) (1991) 25–36.
[7] W. Obal II, W. Sanders, Measure-adaptive state-space construction, Performance Evaluation 44 (1-4) (2001) 237–258.
[8] S. Baarir, S. Haddad, J.-M. Ilié, Exploiting Partial Symmetries in Well-formed nets for the Reachability and the Linear Time Model Checking Problems, in: Proc. of WODES'04, Springer Verlag, Reims - France, 2004.
[9] E. A. Emerson, R. J. Trefler, From Asymmetry to Full Symmetry: New Techniques For Symmetry Reduction in Model Checking, in: Proc. of CHARME'99, LNCS, Springer Verlag, Bad Herrenalb - Germany, 1999, pp. 142–156.
[10] S. Haddad, J. Ilié, K. Ajami, A model checking method for partially symmetric systems, in: Proceedings of FORTE/PSTV'00, Kluwer Academic Publishers, Pisa, Italy, 2000, pp. 121–136.
[11] S. Haddad, J. Ilié, M. Taghelit, B. Zouari, Symbolic Reachability Graph and Partial Symmetries, in: Proc. of the 16^{th} ICATPN, Vol. 935 of LNCS, Springer Verlag, Turin, Italy, 1995, pp. 238–257.
[12] S. Baarir, C. Dutheillet, S. Haddad, J.-M. Ilié, On the use of exact lumpability in partially symmetrical Well-formed Nets, in: Proc. of 2^{nd} Int. Conf. on the Quantitative Evaluation of Systems, IEEE C.S. press, Torino - Italy, 2005, pp. 23–32.
[13] M. Beccuti, S. Baarir, G. Franceschinis, J.-M. Ilié, Efficient lumpability check in partially symmetric systems, in: 3^{rd} Int. Conf. on Quantitative Evaluation of Systems, IEEE Computer Society, Riverside, CA, USA, 2006, pp. 211–221.
[14] S. Baarir, M. Beccuti, D. Cerotti, M. D. Pierro, S. Donatelli, G. Franceschinis, The GreatSPN Tool: Recent Enhancements, ACM Performance Evaluation Review Spec.Issue on Tools for Perf.Eval.
[15] A. S. Miner, G. Ciardo, S. Donatelli, Using the exact state space of a Markov model to compute approximate stationary measures, in: Proc. of the 2000 ACM SIGMETRICS Int. Conf. on Measurement and modeling of computer systems, ACM, Santa Clara, CA, USA, 2000, pp. 207–216.
[16] P. Bazan, R. German, Approximate Analysis of Stochastic Models by Self-Correcting Aggregation, in: Proc. of 2^{nd} Int. Conf. on the Quantitative Evaluation of Systems, IEEE C. S., Torino, Italy, 2005, pp. 134–144.
[17] J. Kemeny, J. Snell, Finite Markov chains, D. Van Nostrand-Reinhold, New York, NY, 1960.
[18] J. Ledoux, Weak lumpability of finite Markov chains and positive invariance of cones, Tech. rep., IRISA (1996).
[19] P. J. Schweitzer, Aggregation methods for large Markov chains, in: Proc. of IWCPR, 1984, pp. 275–286.
[20] R. Paige, R. E. Tarjan, Three partition refinement algorithms, SIAM J. *Comput.* 16 (6) (1987) 973–989.
[21] S. Derisavi, H. Hermanns, W. H. Sanders, Optimal State-Space Lumping in Markov Chains, Information Processing Letters 87 n.6 (6) (2003) 309–315.
[22] A. Valmari, G. Franceschinis, Simple $O(m \log n)$ Time Markov Chain Lumping, in: 16th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Vol. 6015 of LNCS, Springer, Paphos, Cyprus, 2010, pp. 38–52.
[23] B. Plateau, On the Stochastic Structure of Parallelism and Synchronization Models for Distributed Algorithms, in: ACM Sigmetrics Conference on Measrurement and Modelling of Computer Systems, Austin, Texas, 1985.
[24] S. Baarir, M. Beccuti, C. Dutheillet, G. Franceschinis, From partially to fully lumped Markov chains in Stochastic Well Formed Petri Nets, in: 4^{th} Int. Conf. on Performance Methodologies and Tools, ACM Digital Library, Pisa, Italy, 2009.
[25] S. Baarir, M. Beccuti, C. Dutheillet, G. Franceschinis, S. Haddad, Performance analysis of partially symmetric SWNs: efficiency characterization through some case studies , Tech. Rep. TR-INF-2009-07-06-UNIPMN, Dipartimento di Informatica, Università del Piemonte Orientale, http://www.di.unipmn.it/Tecnical-R (2009).