Efficient lumpability check in partially symmetric systems

M. Beccuti[1]

Dip.Informatica, Univ. del Piemonte Orientale, Alessandria, Italy
beccuti@mfn.unipmn.it

1 Introduction

In this extended abstract we are going to present one of the topics of my thesis concerning the performance analysis of stochastic models. During a state space based performance analysis of stochastic models a well known challenging problem is represented by the state space explosion. This problem can be mitigated in systems exhibiting symmetric behavior by aggregating equivalent states and transitions. The Stochastic Well-Formed Colored Nets (SWNs) [1] are an high-level Petri Net formalism which allows automatically to build a quotient graph of the system behavior, called Symbolic Reachability Graph (SRG) [2], where the system symmetries are captured. A lumped Continuous Time Markov Chain (CTMC) [1] with the same size of the SRG can be directly derived from the SRG.

Unfortunately this approach loses most of its efficiency when applied to partially symmetric systems. In these systems the Extended SRG (ESRG) [3] can be used obtaining more state space reduction than using the SRG. In this case it is not always possible to derive directly a Lumped CTMC from the ESRG, because the ESRG may not satisfy the lumpability condition [4] at the CTMC level, hence the ESRG structure might need a refinement to satisfy this condition.

In this extended abstract we present a new efficient algorithm to refine the ESRG with respect to the Exact or to the Strong lumpability condition.

The preliminary results obtained by this algorithm, based on the Paige and Tarjan's partition refinement algorithm [5], will be compared with those presented in the related literature [2,6,7].

We assume the reader is already familiar with the Petri Net notation, both uncolored and colored version [1]. We also assume the reader to be the familiar with the SRG notation [2].

2 Extended Symbolic Reachability Graph

In this section we introduce an intuitive definition of Extended Symbolic Reachability Graph, a more rigorous notation and formal proofs can be found in [8].

We have already introduced that the gain obtained by using the SRG instead of the ordinary RG can be relevant only if the models are highly symmetric, i.e. if they represent a system made of several components of the same type, sharing the same behavior. In the partially symmetrical models, where in general components of the same type have similar behavior but occasionally may behave in different way, the SRG achieves less aggregation. This is due to the need of maintaining a level of detail in the state representation which is adequate when the different behaviors arise, but is redundant elsewhere.

This problem has motivated the introduction of the *Extended Symbolic Reachability Graph (ESRG)*, where the state description detail level is refined only when this is necessary.

The central notion on which the ESRG is based is the *Extended Symbolic Marking* (ESM). An ESM is a two level description: the *Symmetric Representation* level (SR) where the splitting of classes in static subclasses is ignored, and the *eventuality level* where instead it is taken into account.[1]

The eventuality level can be left implicit in all cases where the splitting of classes into subclasses is not necessary to represent the model evolution, and every marking represented by SR corresponds to a reachable state of the model, such that the ESM folds together a number of nodes of the SRG. These ESMs are called *saturated/symmetric*.

In the other cases the eventuality level of an ESM must be explicitly expressed.

An example of this is the ESRG in Fig.2 built from the SWN model in Fig.1. An ESM is represented graphically as a box comprising two parts: in the upper part the SR is shown; in the lower part the the reachable eventualities are shown. All ESMs in the example ESRG are saturated/symmetric except the ESMs $\widehat{\widehat{m}}_{10}$ and $\widehat{\widehat{m}}_{12}$.

According to the double level used in the ESM representation two different types of transition firing are defined: the *generic firing*, where the static subclass partition is ignored, and the *instantiated firing*, where that partition is taken into account. In the example of Fig.2 the generic firings are represented by generic arcs (thick arcs) departing from an SR, while the instantiated firings are represented by instantiated arcs (thin arcs) departing from an eventuality of an ESM.

In Fig.2 the transition instance $(t_5, \langle Z^2 \rangle)$ enabled in $\widehat{\widehat{m}}_{11}$ is generic. Instead, the instance $(t_4, \langle Z^1_{C_2}, Z^1_{C_1} \rangle)$, enabled in the eventuality e_3 of ESM $\widehat{\widehat{m}}_{10}$, is instantiated since it refers to the dynamic subclasses [2] defined into the eventuality.

From the ESRG in Fig.2 we can observe that the asymmetric behavior of model is represented by transition t_4, in particular if all instances of this transition have the same firing rate the aggregation induced by the ESRG satisfies the strong lumpability condition and a lumped CTMC of the same size of the ESRG can be constructed.

[1] Technically speaking this detail level is regulated by grouping similar components into "color classes" and partitioning such color classes into "static subclasses" to be able to distinguish among components with different behaviors

[2] they define a parametric partition of static color subclasses

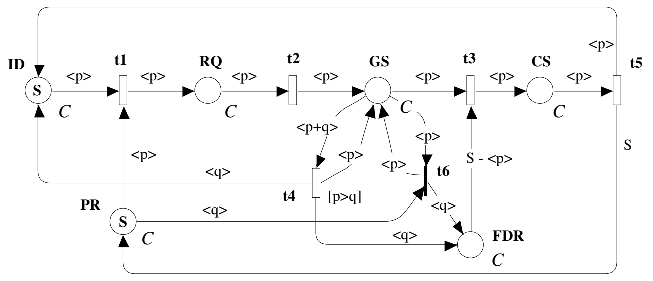

Fig. 1. Petri net model of the Distributed Critical Section algorithm.

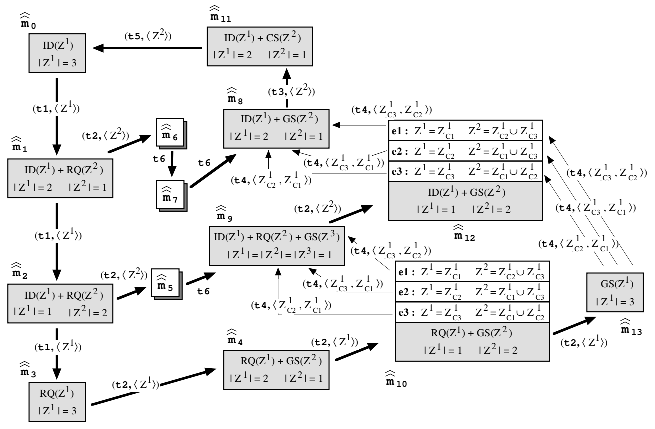

Fig. 2. ESRG of the Petri net model of the DCS algorithm.

If instead the transition t_4 firing instances have different rates, the states in ESM $\widehat{\widehat{m}}_{10}$ do not have the same output rate and the strong lumpability condition is not satisfied, hence the stochastic behavior of the system can not be studied taking the whole ESM as an aggregate. A backward splitting of such aggregate is carried out , whenever the lumpability condition does not hold.

3 Algorithm Description

In this section, we describe an extension of R. Paige and R. Tarjan's partition refinement algorithm [5], for the strong and exact lumpability check of the (SRG) states aggregation induced by the *ESRG*. We are going to present the algorithm for the strong lumpability check that we will extend in order to test also the exact condition.

With respect to R. Paige and R. Tarjan's algorithm this extension uses a different aggregation condition (the strong or the exact lumpability condition) and works using the information contained in the ESRG.

In particular the *stability condition* used in the R. Paige and R. Tarjan's algorithm is weaker than the strong lumpability one. For exact aggregation similar arguments can be appled, but considering dual graph with reversed arcs.[3]

In the next subsection we present an intuitive description of the algorithm for strong lumpability

3.1 Algorithm for Strong Lumpability

Like the R. Paige and R. Tarjan's Partition refinement algorithm [5], our extension uses several data structures, namely Q, X, C described hereafter.

Q is a double-linked list and represents the current partition of symbolic markings[4]; every element of the list is called block [5]. A single block contains a set of elements of type *Node* . A *Node* element can be either a SR or an eventuality of an ESM. For every *Node* element, the following information are stored:1)***type*** : it is set to 'instance' if the *Node* element represents an eventuality of an ESM; to 'macroc' if it represents a saturated symmetric ESM (only the SR part is needed in this case); 2)***in_generic, in_inst*** : are the lists of all input generic/instantiated transition firings which reach this *Node* . Every element in these lists contains a pointer to the source *Node* and the transition rate (weight); 3)***out_generic, out_inst*** : are the lists of all output generic (instantiated) transition firings enabled in this *Node* . Every element in these lists contains a pointer to the destination *Node* and the transition rate (weight).

X is a double-linked list and represents another possible partition of the symbolic markings such that Q is a refinement of X and Q satisfies the lumpability condition with respect to every block of X. Every block in X is not characterized by a list of *Node* elements, but it is described as a list of one or more Q blocks (in the former case it is called *simple block*, in the latter it is called *compound block*). The *Node* elements contained in each X block can be derived as the union of the *Node* elements of the Q blocks contained in X.

C is a list; it is used to maintain pointers to the compound blocks of X (to retrieve them more efficiently).

After the introduction of the data structures we give here an informal description of the algorithm.

It can be divided into two phases: the initialization phase and the iterative refinement phase.

In the first phase, the data structures are initialized. We create the initial Q list using the ESRG : for each saturated symmetric ESM, we shall insert a new block into the Q list which will contain only one element of type 'macroc'. For each other type of ESM, we shall insert a new block into the Q list which will contain as many elements of type 'instance' as the eventualities of this ESM.

[3] It is important to observe that the exact lumpability condition gives the possibility to compute all performance indices computed on the RG. This is not possible using the strong lumpability

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

After that we have to creates the lists X and C and pre-split the aggregates of Q. A Q block will be split iff one or more asymmetric (instantiated) transitions are enabled in it, and the splitting is performed on the basis of the weights and destination ESMs of the outgoing asymmetric firings only. In fact the generic firing are surely symmetric and cannot cause any split. The list X will contain n blocks, where n is the number of the input Q block before the initial pre-split. For every block of Q, that is not split in this step, a simple X block, containing it, is inserted. A compound X block is inserted for every split block of Q, and it will contain all the new Q blocks generated by its splitting. For every compound X block, one block is inserted in C.

At the end of this initialization phase we have obtained the two partitions Q,X, so that Q satisfies the lumpability condition with respect to every block of X.

The second phase consists of repeating the refinement step until $C = \emptyset$. The refinement step works as follows:
a block S is removed from C and a Q block B contained in this X block is selected. After this S is divided in $S' = B$ and $S'' = S - B$. If S'' is compound then we put S'' into C.

The list *e_b* of elements which reach an element in B is built as follows: for every *Node* element in B, we put in *e_b*, the *Node* elements which reach it. These elements are found scanning for every *Node* element in B its lists *in_inst* and *in_generic* and they are inserted in *e_b* only if they are not presented in the list, otherwise only the rate associated with the corresponding element in *e_b* is updated. Note that all macroc elements , before being inserted in *e_b*, will have to be instantiated, so that the macroc elements will be substituted by all the eventualities which are represented by them.

Then all elements in *e_b* are grouped in sublists such that every sublist contains all elements that reach B with the same rate. We split the Q blocks according to these sublists: a generic Q block is split if it contains one or more *Node* elements stored into these sublists. All these *Node* elements are remove from this block and separated into one or more new Q blocks: a separate block for each outgoing rate category (sublists) is created.

At the end of the refinement step the X block containing the split Q block is updated, this Q block must be replaced with the new Q blocks. If this X block was simple then it becomes compound. For every new compound block in X, a new block is inserted in C.

3.2 Algorithm for Exact Lumpability

It is easy to extend the previous algorithm to check the exact lumpability condition. In fact a very short part of the previous algorithm must be modified, since all we need to do is to swap in the previous description the *in_generic* arcs with *out_generic* and the *in_inst* with *out_inst*.

According to this observation the initialization phase must be modified in this way: a Q block will be split iff one or more asymmetric (instantiated) transitions firing instances reach at least one its eventualities. These splitting is performed on the basis of the weights and source ESMs of the only incoming instantiated firings. Like in the previous version for every block of Q that is not split in this step, a simple X block, containing it, is inserted. A compound X block is inserted for every split block of Q, and it will contain all the new Q blocks generated by its splitting. For every compound X block, one block is inserted in C.

In the refinement step only the following change is necessary: after selecting the block B and updating S,X and C , like in the previous version, for every *Node* element in B, we put in the list *e_b* the *Node* elements which this element reaches. These *Node* elements are found scanning for every *Node* element in B its lists *out_inst* and *out_generic*. Like in the strong lumpability check version the elements are inserted in *e_b*, if they have not yet been added, otherwise only the corresponding rate is updated. All macroc elements found, before inserting in *e_b*, will have to be instantiated.

Then all elements in *e_b* are grouped into sublists such that every sublist contains those elements that B reaches with the same rate.

After this the refinement step works in the same way of the previous version.

4 Some Results and Discussion

In this section we are going to show that the efficiency of the approach presented in this paper relies on two main factors : the fact that it is based on Paige and Tarjan's algorithm and that it exploits the ESRG as a starting point. We will compare this algorithm with:

– the algorithms applying the Paige and Tarjan's algorithm on the SRG without any knowledge of the ESRG and working only on the strong lumpability condition;
– the algorithm proposed in [8] based on the ESRG and working only on the strong lumpability condition;
– the algorithm presented in [7] based on Dynamic SRG (DSRG) which satisfies the exact lumpability condition by construction.

The comparison between our algorithm and those applying the Paige and Tarjan's algorithm on the SRG, will be made in terms of number of steps (which gives a measure of the time saving) and of the number of not instantiated ESMs (which gives a measure of the space saving). This is a fair comparison, even if in our algorithm the single step may require, before checking the lumpability condition, the instantiation of the SMs not explicitly represented in an ESM. In fact our algorithm calls a function that instantiates the SMs not yet explicitly represented in the ESM, but since this function is called only one time for every ESM and its (worst case) cost is constant we can consider again the split steps comparable, in terms of order of magnitude complexity. Finally, the cost of the instantiation function, would anyway be paid in the SRG generation phase in case the ESRG was not used. In practice for the comparison purpose we have realized a modified version of our algorithm, where X initially contains only one block comprising all blocks of Q and where all the eventualities are instantiated.[5]

This modified algorithm in fact mimics the behavior of other algorithms based on Paige and Tarjan's one, and differs from those algorithms only because it starts with an initial refinement that prevents to reach an aggregation that is coarser than that induced by the ESRG, so that the number of refinement steps of the modified algorithm is a lower bound with respect to these other algorithms ([6,4,9,10]).

Although in some (rare) cases it may happened that a coarser partition than that induced by the ESRG could be reached by the other algorithms, this is not desirable, because it could prevent the computation of some relevant performance indices.

#Proc.	#SRG	#ESRG	#Refined (strong)	#Steps X=ESRG	#Steps X=SRG	#Non split ESMS	#DSRG	#Refine (exact)	#Peak
3	45	14	21	8	16	2(9)	23	21	27
4	441	20	41	20	33	3(28)	49	40	94
7	3704	26	58	26	53	6(400)	83	63	237
9	28159	40	89	59	88	18(2969)	125	90	367

Table 1. Results computed on the DCS model (using strong lumpability)

In Table 1 (columns 1-7) are summarized the results computed on the DCS model (Fig.2), where the first column shows the number of processes that compete for entering in the critical section, the second, the third and the fourth columns contain the number of states of the SRG, ESRG and refined ESRG; the fifth and the sixth columns contain the number of algorithm steps with initially X={one block with all SRG states} or with X=ESRG. The seventh column contains the number of non split ESMs (macrocs) in the refined ESRG, and also includes the number of corresponding (never instantiated) eventualities.

We can observe that our algorithm obtains the refined ESRG in less steps and using less memory (the final ESRG still contains some not instantiated saturate symmetric ESMs).

A direct comparison with the previous ESRG-based algorithm proposed in [8] is not possible, because no implementation of such algorithm is available. However we can observe that the use of Paige and Tarjan's algorithm has allowed us to build an algorithm with complexity $O(m\ log(n))$, (where n is the number of SRG nodes, and m is the number of transitions between the nodes in the corresponding CTMC) which is better than the $O(n\ m)$ complexity of the algorithm in [8].

[5] In other words there are no *macroc* elements, so that the required memory space is usually greater than that of the algorithm proposed in this paper

The previous comparisons are related only to the strong lumpability version of our algorithm, now we want to compare the exact lumpability check extension with the algorithm proposed in [7]. The two approaches have some relevant differences, since the latter is based on the computation of the so-called Dynamic SRG (DSRG) which satisfies the exact lumpability condition by construction. In the DSRG based approach it may happen that some markings (corresponding to the SMs) are replicated in the final lumpable structure, so it can be happened that it may contain more aggregates than the refined ESRG (it would interesting to find a characterization of those classes of models that can gain more from the use of a one aggregation algorithm with respect to the alternative ones). We must also consider that in case several analysis have to be performed on the same model while varying the rate of same transition in our algorithm it is sufficient to run the lumpability check once more, reusing the previously constructed ESRG, while in the DSRG based approach, any variation in the model requires to start the construction of the DSRG all over again.

The last three columns in Table 1 show the results computed on the DCS model (Fig.2) where in the eighth and ninth columns contain the number of states of the DSRG and refined ESRG. The tenth column contains the real number of ESMs and eventualities stored during the computation.

We can observe that for this model the final aggregation obtained by our algorithm wins, but that the number of real ESMs and eventualities stored during the computation (i.e. the peak memory occupation) is greater than of the DSRG. In fact our algorithm conserves all eventualities in a Q block until the end of refinement, because a Q block can be visited more that one time and rebuilding them every time it is too expensive.

Acknowledgments

This work is carried out under the supervision of the Professors G. Franceschinis[6] and S. Haddad [7]

References

1. G. Chiola, C. Dutheillet, G. Franceschinis, S. Haddad: Stochastic Well-formed Coloured nets for symmetric modelling applications. IEEE Transactions on Computers **42** (1993) (11): 1343 – 1360
2. G. Chiola, C. Dutheillet, G. Franceschinis, S. Haddad: A Symbolic Reachability Graph for Coloured Petri Nets. Theoretical Computer Science B (Logic, semantics and theory of programming) **176** (1997) (1,2): 39 – 65
3. S.Haddad, J.M Ilié, M. Taghelit, B. Zouari: Symbolic Reachability Graph and Partial Symmetries. Sixteenth International Conference on Application and Theory of Petri Nets. Turin, Italy. June 1995. Lecture Note in Computer Science **935** (1995) 238–251

[6] Dip.Informatica, Univ. del Piemonte Orientale, Alessandria, Italy
[7] LAMSADE-CNRS UMR 7024, Universitè Paris Dauphine, Paris, France

4. P. Buchholz: Efficient computation of equivalent and reduced representations for stochastic automata. Internat. J. Comput. Systems Sci. Engrg. **15(2)** (2000) 93–103
5. R. Paige, R. E. Tarjan: Three partition refinement algorithms. SIAM J. *Comput.* **16 (6)** (1987) 973–989
6. S. Derisavi, H. Hermanns, W. H. Sanders: Optimal State-Space Lumping in Markov Chains. Information Processing Letters **87 n.6** (2003) 309–315
7. Baarir S., Dutheillet C., Haddad S., Ilié J-M.: On the use of exact lumpability in partially symmetrical well-formed nets. In Society, I.C., ed.: proceedings of the 2nd IEEE international Conference on the Quantitative Evaluation of Systems, Torino - Italy, IEEE Computer Society (2005) 23–32 best paper award.
8. L. Capra, C. Dutheillet, G. Franceschinis, J-M. Ilié: Exploiting Partial Symmetries for Markov Chain Aggregation. Electronic Notes In Theoretical Computer Science **39 (3)** (2000)
9. M. Bernardo, R. Gorrieri: A tutorial on empa: a theory of concurrent processes with nondeterminism, priorities, probabilities and time. Theor. Comput. Sci. **202** (1998) 1–54
10. H. Hermanns, M. Siegle: Bisimulation algorithms for stochastic process algebras and their bdd-based implementation. In: ARTS '99: Proceedings of the 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems, London, UK, Springer-Verlag (1999) 244–264

A Algorithm Description for Strong Lumpability

The pseudo-code of the algorithm is given in Fig.3 and it can be divided into two phases: the initialization [lines 1-2] and the iterative refinement [lines 3-30].

Create_X_C(): will split a Q block iff one or more asymmetric (instantiated) transitions firing instances reach at least one its eventuality, and the splitting is performed on the basis of the weights and source ESMs of the only outgoing instantiated firings. For every block of Q, that is not split in this step, a simple X block, containing it, is inserted. A compound X block is inserted for every split block of Q, and it will contain all the new Q blocks generated by its splitting. For every compound X block, one block is inserted in C.

Compute_E(): for every *Node* element in B, put in the list *e_b* the *Node* elements which reach this B element. These *Node* elements are found scanning for every *Node* element in B its lists *in_inst* and *in_generic*. They are inserted in *e_b*, if they have not yet been added, otherwise only the corresponding rate is updated. All macroc elements found, before inserting in *e_b*, will have to be instantiated.

Prepare_split(): will build the list of lists *l_list* where every sublist $sublist_k$ contains all elements that reach B with the same rate.

Algorithm

```
1:  Create_Q(ESRG);
2:  C=Create_X_C(Q);
3:  while (C!=NULL) do
4:     S=C.top(); {remove a compound aggregate S from C}
5:     B=S.find_new_b(); {find a sub-aggregate of S which is a block in Q}
6:     build S'=B e S"=S-B;
7:     if (S".is_compound()) then
8:        C.push(S");
9:     end if
10:    e_b=Compute_E(B); {compute the set of blocks that reach B}
11:    l_list=Prepare_split(e_b); {partition the elements of e_b according to their
       outgoing rate}
12:    for (sublist_k ∈ l_list) do
13:       for (x_j ∈ sublist_k) do
14:          new_q_block = ∅; {initialize the list of new Q blocks}
15:          D_i = x_j.find_set(); {find the block D_i which contains it}
16:          D_i.remove(x_j);
17:          block=new_q_block.search(D_i);{return the block corresponding to D_i
             in new_q_block or NULL if it does not exist}
18:          if (block==NULL) then
19:             block=new_q_block.create(D_i);{create the block corresponding to D_i
                in new_q_block}
20:          end if
21:          block.insert(x_j);
22:       end for
23:       while (nblock=new_q_block.top()!=NULL) do
24:          block=Q.insert(nblock);{return a pointer to the inserted block}
25:          Xblock=block.find_blockX();
26:          Xblock.update(block);
27:          if (Xblock().is_compound()) then
28:             C.push(Xblock);
29:          end if
30:       end while
31:    end for
32: end while
33: return Q;
    Note: the queue.top() method removes and returns the first element in queue.
```

Fig. 3. Algorithm for the *ESRG* lumpability check

B Algorithm for Exact Lumpability

It is easy to extend the previous algorithm in order to check the exact lumpability. In order to extend the previous algorithm for checking the exact lumpability only the functions: *Create_X_C*(),*Compute_E*(), and *Prepare_split*() have to be modified.

Create_X_C(): will split a Q block iff one or more asymmetric (instantiated) transitions firing instances reach at least one its eventuality, and the splitting is performed on the basis of the weights and source ESMs of the only incoming instantiated firings. Like in the previous version for every block of Q, that is not split in this step, a simple X block, containing it, is inserted. A compound X block is inserted for every split block of Q, and it will contain all the new Q blocks generated by its splitting. For every compound X block, one block is inserted in C.

Compute_E(): for every *Node* element in B, put in the list *e_b* the *Node* elements which this element reaches. These *Node* elements are found scanning for every *Node* element in B its lists *out_inst* and *out_generic*. Like in the previous version they are inserted in *e_b*, if they have not yet been added, otherwise only the corresponding rate is updated. All macroc elements found, before inserting in *e_b*, will have to be instantiated.

Prepare_split(): will build the list of lists *l_list* where every sublist $sublist_k$ contains all elements that B reaches with the same rate.