[10] G. Ciardo and K. S. Trivedi. A decomposition approach for stochastic reward net models. *Perf. Eval.*, 18(1):37–59, 1993.

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* The MIT Press, Cambridge, MA, 1990.

[12] S. Donatelli. Superposed generalized stochastic Petri nets: definition and efficient solution. In R. Valette, editor, *Application and Theory of Petri Nets 1994, Lecture Notes in Computer Science 815 (Proc. 15th Int. Conf. on Applications and Theory of Petri Nets, Zaragoza, Spain)*, pages 258–277. Springer-Verlag, June 1994.

[13] P. Kemper. Numerical analysis of superposed GSPNs. In *Proc. Int. Workshop on Petri Nets and Performance Models (PNPM'95)*, pages 52–61, Durham, NC, Oct. 1995. IEEE Comp. Soc. Press.

[14] D. E. Knuth. *Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

[15] B. Plateau and K. Atif. Stochastic Automata Network for modeling parallel systems. *IEEE Trans. Softw. Eng.*, 17(10):1093–1108, Oct. 1991.

[16] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains.* Princeton University Press, 1994.

[17] M. Tilgner, Y. Takahashi, and G. Ciardo. SNS 1.0: Synchronized Network Solver. In *1st International Workshop on Manufacturing and Petri Nets*, pages 215–234, Osaka, Japan, June 1996.

A further technique based on event locality achieves an additional reduction in the execution time, with no additional memory cost.

The results substantially increase the size of the reachability sets that can be managed, and they can be particularly effective for the Kronecker-based approaches that have recently been proposed.

In the future, we will investigate how to use our approach in a distributed fashion, as done in [6], where N cooperating processes explore different portions of the reachability set. In particular, we plan to apply the data structure we presented and to explore ways to balance the load among the cooperating processes automatically.

References

[1] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with generalized stochastic Petri nets*. John Wiley & Sons, 1995.

[2] P. Buchholz. Numerical solution methods based on structured descriptions of Markovian models. In G. Balbo and G. Serazzi, editors, *Computer performance evaluation*, pages 251–267. Elsevier Science Publishers B.V. (North-Holland), 1991.

[3] P. Buchholz and P. Kemper. Numerical analysis of stochastic marked graphs. In *Proc. Int. Workshop on Petri Nets and Performance Models (PNPM'95)*, pages 32–41, Durham, NC, Oct. 1995. IEEE Comp. Soc. Press.

[4] G. Chiola. Compiling techniques for the analysis of stochastic Petri nets. In *Proc. 4th Int. Conf. on Modelling Techniques and Tools for Performance Evaluation*, pages 13–27, 1989.

[5] G. Ciardo, A. Blakemore, P. F. J. Chimento, J. K. Muppala, and K. S. Trivedi. Automated generation and analysis of Markov reward models using Stochastic Reward Nets. In C. Meyer and R. J. Plemmons, editors, *Linear Algebra, Markov Chains, and Queueing Models*, volume 48 of *IMA Volumes in Mathematics and its Applications*, pages 145–191. Springer-Verlag, 1993.

[6] G. Ciardo, J. Gluckman, and D. Nicol. Distributed state-space generation of discrete-state stochastic models. *ORSA J. Comp.* To appear.

[7] G. Ciardo and A. S. Miner. SMART: Simulation and Markovian Analyzer for Reliability and Timing. In *Proc. IEEE International Computer Performance and Dependability Symposium (IPDS'96)*, page 60, Urbana-Champaign, IL, USA, Sept. 1996. IEEE Comp. Soc. Press.

[8] G. Ciardo, J. K. Muppala, and K. S. Trivedi. On the solution of GSPN reward models. *Perf. Eval.*, 12(4):237–253, 1991.

[9] G. Ciardo and M. Tilgner. On the use of Kronecker operators for the solution of generalized stochastic Petri nets. *J. of the Association for Computing Machinery*. Submitted.

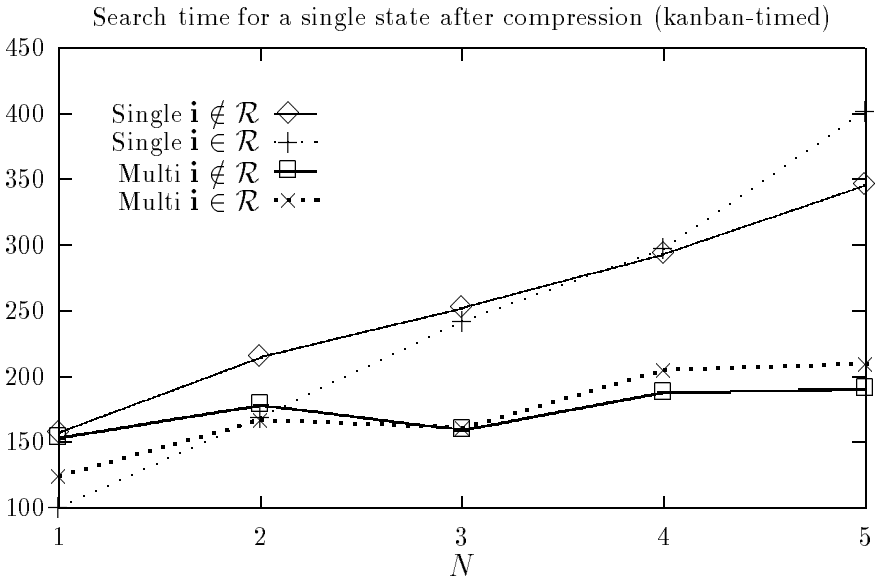

Figure 14: Expected search times (microseconds) for the single and multilevel approach.

the main concern, is greatly reduced. At the same time, this reduction is achieved not at the expense of, but in conjunction with, execution efficiency.

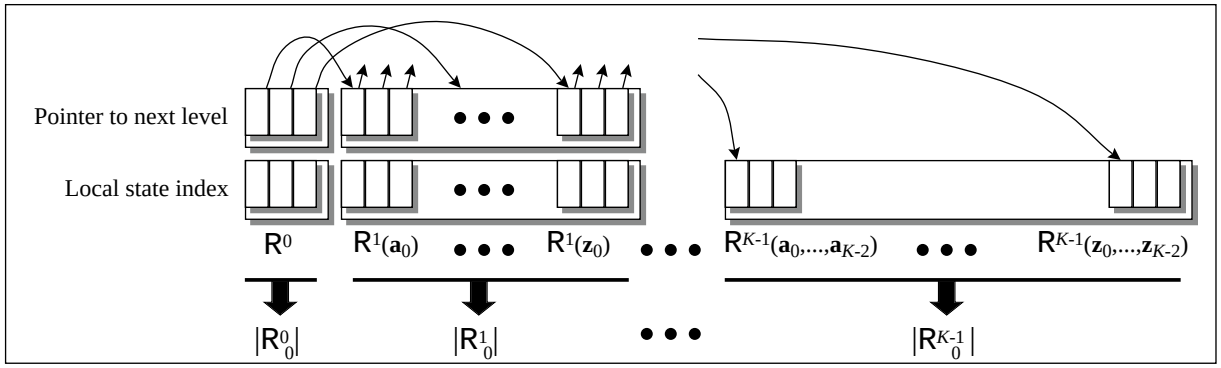

Figure 13: Multilevel storage after the reachability set has been built.

desirable, since, as discussed before, only the last level has a major effect on the memory requirements. Using 8 or 16 bit indices, we can then store $\mathcal{R}$ using only slightly more than $|\mathcal{R}|$ or $2|\mathcal{R}|$ bytes, respectively.

The execution time required to transform our tree-based multilevel data structure into the array-based one is negligible, around 1/1000 of the time used by *BuildRS*. Hence, this transformation is always advisable if searches are to be performed on $\mathcal{R}$. This is the case, for example, when computing $\Psi(\mathbf{i})$ for some $\mathbf{i} \in \hat{\mathcal{R}}$, one of the essential operations in the Kronecker approach we discussed. To further investigate the impact of our multilevel data structure, we computed the expected time required to search for a reachable state $\mathbf{i} \in \mathcal{R}$ (i.e., to compute $\Psi(\mathbf{i})$ when $\mathbf{i} \in \mathcal{R}$), and for a non-reachable state (i.e., to compute $\Psi(\mathbf{i})$ when it is null) in the kanban-timed model. Fig. 14 shows the results for the single vs. the multilevel approach. The dramatic difference in the slope of the curves for the single and multilevel approaches further confirms that our results will greatly improve the efficiency of solution methods based on Kronecker operators.

Also, note how the curves for $\mathbf{i} \in \mathcal{R}$ and $\mathbf{i} \notin \mathcal{R}$ intersect, in both cases. We generated unreachable states by determining the bound b_p on the number of tokens in each place p, then randomly choosing a number n_p of tokens between 0 and b_p for p, independently of the other places. The resulting marking was (almost) always unreachable. For small reachability sets, searching for a reachable state is faster than searching an unreachable state, because we can sometimes stop the search before reaching a leaf. However, as the state space grows, it is increasingly likely that, when searching for an unreachable state, the comparison between two states (or substates, for the multilevel approach) stops after comparing just a few places. Clearly, this effect more than offsets the need to explore up to a leaf for the single-level approach (in the multilevel approach, an additional advantage is that, for an unreachable state, the search might stop at a level $k < K - 1$).

8 Conclusion

We have presented a detailed analysis of a multilevel data structure for storing and searching the large set of states reachable in some high-level model. Memory usage, which is normally

```
BtrVectMatrMultiply(in: x, K, A^0, A^1, ..., A^{K-1}; inout: y)
 1. for each i_0 ∈ LocalSet(0, null) do
 2.     I_0 ← LocalIndex(0, null, i_0);
 3.     for each j_0 s.t. A^0_{i_0,j_0} > 0 do
 4.         J_0 ← LocalIndex(0, null, j_0);
 5.         if J_0 ≠ null then
 6.             a_0 ← A^0_{i_0,j_0};
 7.             for each i_1 ∈ LocalSet(1, I_0) do
 8.                 I_1 ← LocalIndex(1, I_0, i_1);
 9.                 for each j_1 s.t. A^1_{i_1,j_1} > 0 do
10.                     J_1 ← LocalIndex(1, J_0, j_1);
11.                     if J_1 ≠ null then
12.                         a_1 ← a_0 · A^1_{i_1,j_1};
    ...
13.                         for each i_{K-1} ∈ LocalSet(K − 1, I_{K-2}) do
14.                             I_{K-1} ← LocalIndex(K − 1, I_{K-2}, i_{K-1});
15.                             for each j_{K-1} s.t. A^{K-1}_{i_{K-1},j_{K-1}} > 0 do
16.                                 J_{K-1} ← LocalIndex(K − 1, J_{K-2}, j_{K-1});
17.                                 if J_{K-1} ≠ null then
18.                                     a_{K-1} ← a_{K-2} · A^{K-1}_{i_{K-1},j_{K-1}};
19.                                     y_{J_{K-1}} ← y_{J_{K-1}} + x_{I_{K-1}} · a_{K-1};
```

Figure 12: Improved procedure to multiply a vector by a submatrix of a Kronecker product.

7 Compressing $\mathcal{R}$ after exploration

So far we have discussed techniques to store the reachability set while it is being built by procedure *BuildRS*. After this phase, though, dynamic data structures such as search trees are unnecessary; a much more memory-efficient data structure can be used. For superposed GSPNs, this was proposed by Kemper [13], who uses an integer array of size $|\mathcal{R}|$. The I-th position of this array stores the I-th marking in lexical order, $\Psi^{-1}(I)$. Using this array, a simple binary search can be used to compute $\Psi(\mathbf{i})$ for a given $\mathbf{i} \in \hat{\mathcal{R}}$, where $\Psi(\mathbf{i}) = \mathsf{null}$ if $\mathbf{i} \notin \mathcal{R}$. An underlying assumption in [13] is that each marking can be encoded into a 32-bit integer, and this might not be easy for very large models.

Our multilevel approach can help here as well. We can store the nodes of a tree at level $k < K-1$ as an ordered array of pairs of pointers (see Fig. 13). The first one points to the local state for level k (alternatively, we could store either an index into an array containing the elements of $\mathcal{R}^k$ in any order, or the local state itself); the lexical position of the particular local state determines the array order. The second pointer points to the array used to store the tree at the next level. At the last level, we do not need to store a pointer to a lower-level tree, hence only one pointer or, better, one index is required for each node. This is very

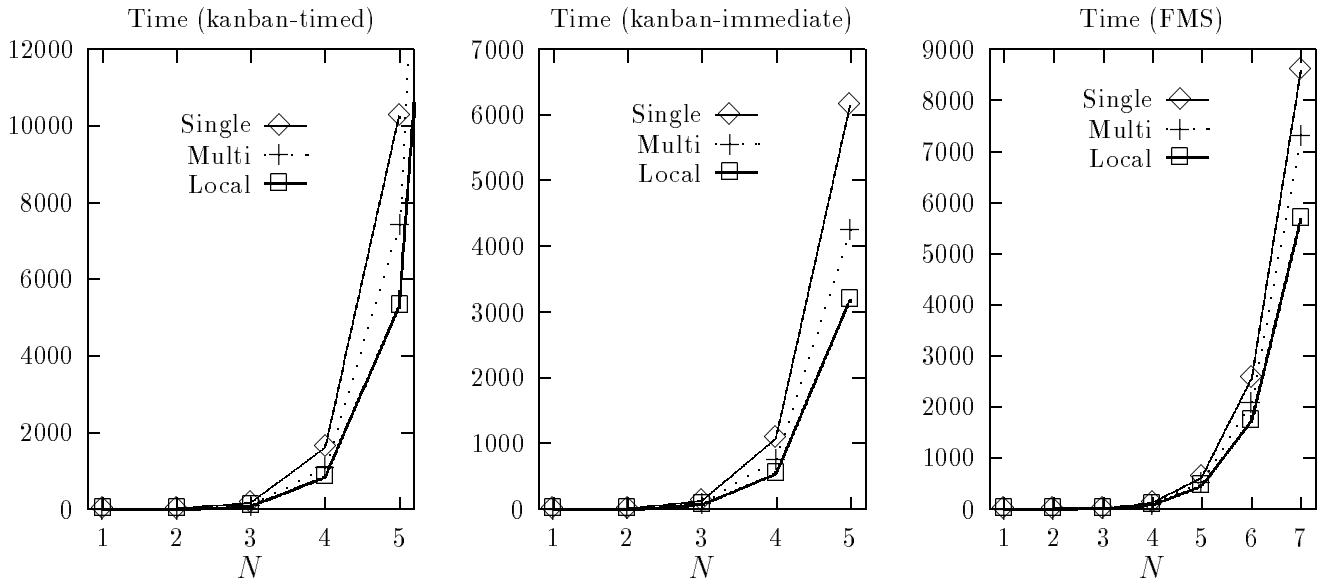

Figure 11: Execution times (seconds) for the single and multilevel approach.

that its components 0 through $k-1$ are identical to those of $\mathbf{i}$. Hence, we can search for it starting at I_{k-1}, that is, we only need to perform the $K-k$ calls

$$LocalIndex(K-1, \ldots LocalIndex(k+1, LocalIndex(k, I_{k-1}, \mathbf{i}_k), \mathbf{i}_{k+1}), \ldots, \mathbf{i}_{K-1}).$$

The savings due to these accelerated searches depend on the particular model and the partition chosen for it. We show the results for our three models in Fig. 11. In this case, savings of 20% execution time or more are achieved. Except for the need to specify a partition, which needs to be done anyway to use our multilevel approach, the technique is completely automatic, and has no additional memory requirements.

6.2 Application to solutions based on Kronecker operators

We can also apply our multilevel approach and the principle of locality just introduced to speed up procedure *VectMatrMultiply*, described in Section 4. An improved version, *BtrVectMatrMultiply* of the same procedure is shown in Fig. 12. Its complexity is

$$\begin{aligned} &O(\eta((\mathbf{A}^0)_{\mathcal{R}_0^0,\mathcal{R}^0}) \cdot \log n_0 + \eta((\mathbf{A}^0 \otimes \mathbf{A}^1)_{\mathcal{R}_0^1,\mathcal{R}_0^0\times\mathcal{R}^1}) \cdot \log n_1 \\ &+ \cdots \eta((\mathbf{A}^0 \otimes \ \cdots \otimes \mathbf{A}^{K-1})_{\mathcal{R}_0^{K-1},\mathcal{R}_0^{K-2}\times\mathcal{R}^{K-1}}) \cdot \log n_{K-1}) \\ &= O(\eta(\mathbf{A}_{\mathcal{R},\mathcal{R}_0^{K-2}\times\mathcal{R}^{K-1}}) \cdot \log n_{K-1}). \end{aligned}$$

Indeed, the term "$\log n_k$" for the searches at level k is an upper bound, since the size of the tree searched by *LocalIndex* is smaller than n_k if there are any unreachable states; the "average" size of a tree at level k is $|\mathcal{R}_0^k|/|\mathcal{R}_0^{k-1}| \leq n_k$.

Section 7 further investigates the complexity of searching a state, reachable or not, after the entire reachability set has been explored.

6.1 Exploiting event locality

Our multilevel search tree has an additional advantage which can be exploited to further speed up execution. To illustrate the technique, we define two functions:

- *LocalSet*(k, I_{k-1}), which, given a submodel index k, $0 \leq k < K$, and a pointer I_{k-1} to a reachable substate $(\mathbf{i}_0, \ldots, \mathbf{i}_{k-1}) \in \mathcal{R}_0^{k-1}$, returns a pointer to the tree containing the set $\mathcal{R}^k(\mathbf{i}_0, \ldots, \mathbf{i}_{k-1})$.

- *LocalIndex*$(k, I_{k-1}, \mathbf{i}_k)$, which, given a submodel index k, $0 \leq k < K$, a pointer I_{k-1} to a a reachable substate $(\mathbf{i}_0, \ldots, \mathbf{i}_{k-1}) \in \mathcal{R}_0^{k-1}$, and a local state index $\mathbf{i}_k \in \mathcal{R}^k$, returns the pointer I_k to substate $(\mathbf{i}_0, \ldots, \mathbf{i}_k)$, if reachable, null otherwise.

Given our data structure, "a pointer I_k to a reachable substate $(\mathbf{i}_0, \ldots, \mathbf{i}_k) \in \mathcal{R}_0^k$" points to the node corresponding to the local state $\mathbf{i}_k$ in the tree containing the set $\mathcal{R}^k(\mathbf{i}_0, \ldots, \mathbf{i}_{k-1})$.

To find whether state $\mathbf{i}$ is in (the current) $\mathcal{R}$, we can then use a sequence of K function calls

$$LocalIndex(K-1, \underbrace{LocalIndex(K-2, \ldots LocalIndex(1, LocalIndex(0, \mathsf{null}, \mathbf{i}_0), \mathbf{i}_1), \ldots \mathbf{i}_{K-2})}_{I_{K-2}}, \mathbf{i}_{K-1}).$$

The overall cost of these K calls is exactly what we just discussed when comparing the single and multilevel approaches. However, if $\mathbf{i}$ is reachable and we now wanted to find the index of a state $\mathbf{i}'$ differing from $\mathbf{i}$ only in its last position, we could do so with a single call *LocalIndex*$(K-1, I_{K-2}, \mathbf{i}'_{K-1})$, provided we have saved the value I_{K-2}, the index of the reachable substate $(\mathbf{i}_0, \ldots, \mathbf{i}_{K-2})$. A similar argument applies to other values of k. Only for $k = 0$ we need to perform an entirely new search.

We can then exploit this "locality" by considering the possible effects of the events on the state. Given $\hat{\mathcal{R}} = \mathcal{R}^0 \times \cdots \mathcal{R}^{K-1}$, we can partition $\mathcal{E}$ into $\mathcal{E}^0, \ldots, \mathcal{E}^{K-1}$, such that

$$e \in \mathcal{E}^k \iff \left(\forall \mathbf{i} \in \hat{\mathcal{R}}, Active(e, \mathbf{i}) = True \wedge \mathbf{j} = New(e, \mathbf{i}) \Rightarrow \forall l, 0 \leq l < k, \mathbf{i}_l = \mathbf{j}_l\right),$$

that is, events in $\mathcal{E}^k$ can only change local states k or higher. For example, for GSPNs, this is achieved by assigning to $\mathcal{E}^k$ all the transitions whose enabling and firing effect on the state depends only on places in sub-GSPN k, and possibly in sub-GSPNs $k+1$ through $K-1$, but not in sub-GSPNs 1 through $k-1$. This can be easily accomplished through an automatic inspection even in the presence of guards and inhibitor and variable cardinality arcs (although in this case the partition might be overly conservative, that is, it might assign a transition to a level k when it could have been assigned to a level $l > k$).

When exploring state $\mathbf{i}$, in procedure *BuildRS*, we remove $\mathbf{i}$ from $\mathcal{U}$ and insert it into $\mathcal{R}$, obtaining, as a byproduct, a sequence of pointers $I_{-1} = \mathsf{null}, I_0, \ldots, I_{K-2}$, to the reachable substates $(\mathbf{i}_0), \ldots, (\mathbf{i}_0, \ldots, \mathbf{i}_{K-2})$. Then, we can examine the events in $\mathcal{E}$ in any order. As soon as we find an enabled event $e \in \mathcal{E}^k$, we generate $\mathbf{i}' = New(e, \mathbf{i})$, and we are guaranteed

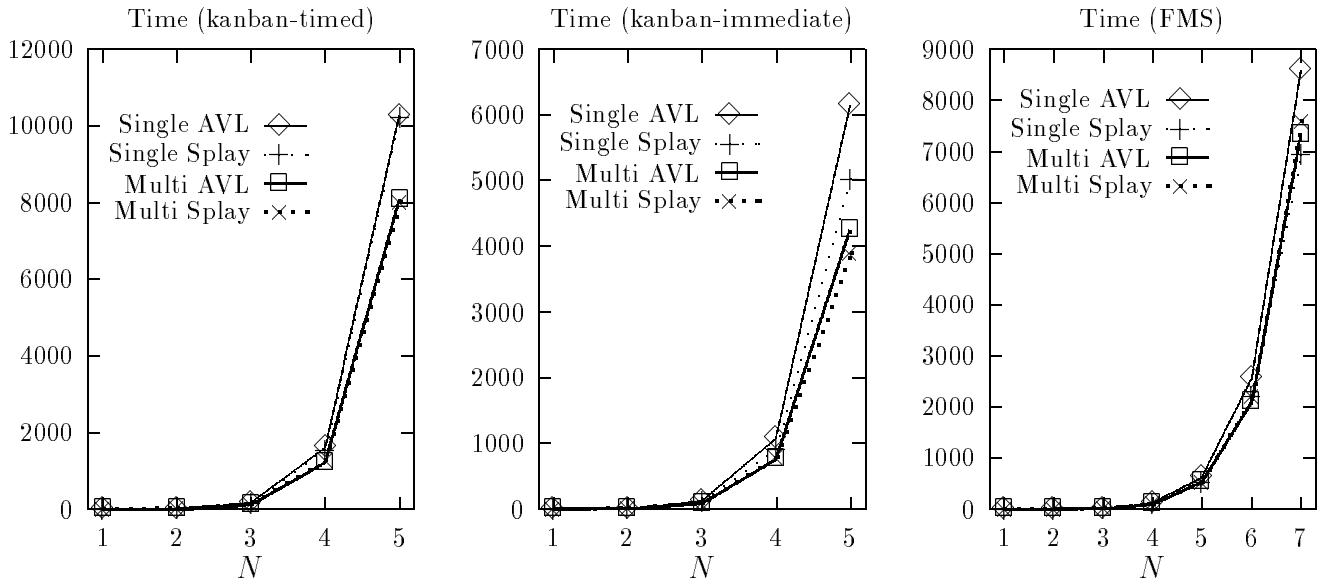

Figure 10: Execution times (seconds) for the AVL and splay trees.

the substate $\mathbf{i}_l$ we are looking for, so the jump from level l to level $l+1$ might occur before reaching a leaf.

- If $\mathbf{i}$ is already in $\mathcal{R}$, instead, the search on the single-level tree will sometimes find $\mathbf{i}$ before reaching a leaf, but this is true also at each level of the multilevel approach, just as in the previous case, for the levels $l < k$.

- Perhaps even more important, regardless of whether $\mathbf{i}$ is already in the tree or not, is the complexity of each comparison. With the multilevel approach, only substates are compared, and these are normally small data structures. For GSPNs, this could be an array of 8 or 16 bit integers, one for each place in the sub-GSPN (memory usage is not an issue, since each local state for level k is stored only once). On the other hand, with the single-level approach, entire states are compared. If the states are stored as arrays, the comparison can stop as soon as one component differs in the two arrays. However, as the search progresses down the tree, we compare states that are increasingly close in lexical order, hence likely to have the same first few components; this implies that comparisons further away from the root tend to be more expensive. Furthermore, storing states as full arrays is seldom advisable. For example, in SMART, a sophisticated packing is used, on a state-by-state base, to reduce the memory requirements. In this case, the state in the node needs to be "unpacked" before comparing it with $\mathbf{i}$, thus effectively examining every component of the state.

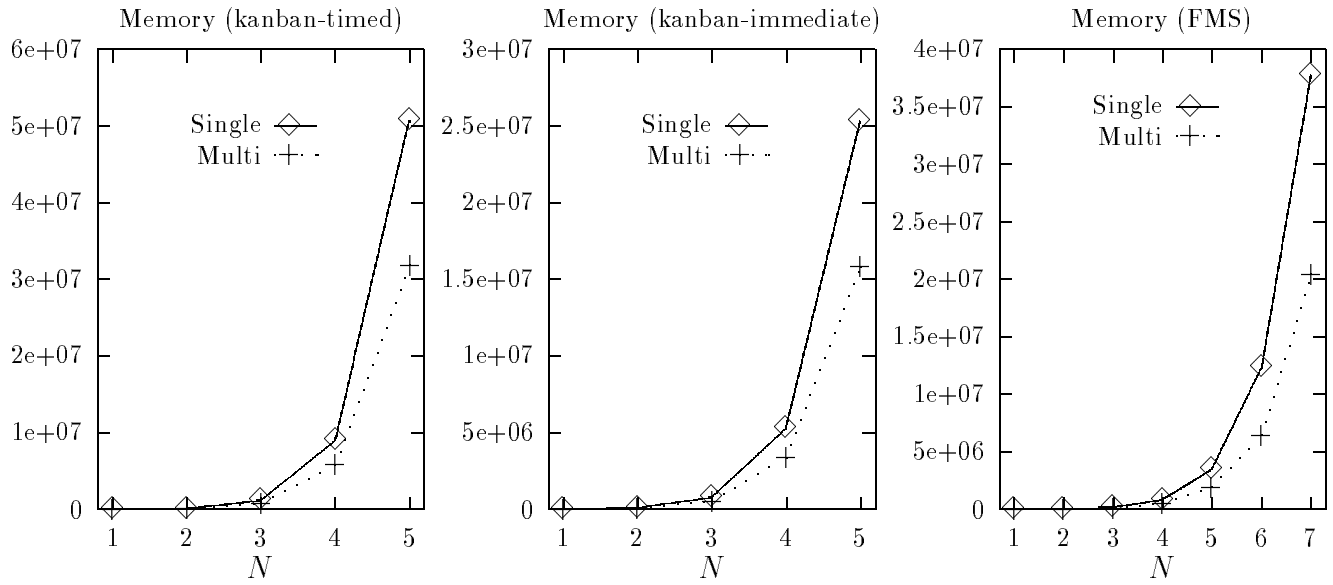

Figure 9: Memory usage (bytes) for the single and multilevel approaches.

Fig. 9 shows the number of bytes used to store $\mathcal{R}$ and $\mathcal{U}$, separately, for our three models, using the single vs. the multilevel approach. The dashed boxes in Fig. 1 and 2 indicate how the models have been decomposed into submodels, in all cases $K = 4$.

The multilevel approach is clearly preferable, especially for large models. Indeed, we could not generate the state space using the single-level approach for $N = 6$ for the kanban-timed model, while we could using the multilevel approach (all our experiments where run on a Sun-clone with a 55 MhZ HyperSparc processor and 128 Mbyte of RAM, without making use of virtual memory).

6 Execution time

Several factors need to be considered when studying the impact of our approach on the execution time. First, we explored possible differences due to using splay trees vs. AVL trees. Fig. 10 shows that no method is clearly better and that, in most cases, the differences are minor in comparison to the effect due to the choice of a single vs. a multilevel data structure.

We then focussed on this second factor, using AVL trees exclusively. The advantages of the multilevel approach are:

- Consider first the case when *BuildRS* searches for a state $\mathbf{i}$ not yet in the current $\mathcal{R}$ (the same discussion holds for $\mathcal{U}$). With a single-level tree, the search will stop only after reaching a leaf, hence $O(\log|\mathcal{R}|)$ comparisons are always performed. In the multilevel approach, instead, the search stops as soon as the substate $(\mathbf{i}_1, \ldots, \mathbf{i}_k)$ is not reachable, for some $k \leq K-1$. If all the trees at a given level are of similar size, this will require at most $O(\log|\mathcal{R}_0^k|)$ comparisons. In practice, the situation is even more favorable, because, for any level $l < k$, the tree searched, which stores $\mathcal{R}^l(\mathbf{i}_1, \ldots, \mathbf{i}_{l-1})$, contains

- First, the total number of nodes at levels 0 through $K-2$ is a small fraction of the nodes at level $K-1$, that is, of $|\mathcal{R}|$, provided the trees at the last level, $K-1$, are not "too small". This is ensured by an appropriate partitioning of a large model into submodels. In other words, only the memory used by the nodes of the trees at the last level really matters.

- Second, the nodes of the trees at the last level do not require a pointer to a tree at a lower level, hence each node requires only three pointers (96 bits). Again, if we can assume a bound on the number of local states for the last submodel, $|\mathcal{R}^{K-1}| \leq 2^b$, we can in principle implement dynamic data structures that require only $3b$ bits per node, plus some overhead to implement dynamic arrays. Given a model, there is usually some freedom in choosing the number of submodels, that is, the number of levels K. We might then be able to define the last submodel in such a way that is has at most 2^8 local states, resulting in only slightly more than $3|\mathcal{R}|$ bytes to store the entire reachability set. When this is not the case, the next easily implementable step, 2^{16}, is normally more than enough. Even in this case, the total memory requirements are still much better than with a single-level implementation.

We stress that, while this discussion about using 8 or 16 bit indices seems to be excessively implementation-dependent, it is not. Only through the multilevel approach it is possible to exploit the small size of each local space. In the single-level approach, instead, the total number of states that can be indexed (pointed by a pointer) is $|\mathcal{R}|$, a number which can be easily be 10^7, and possibly more. In this case, we are forced to use 32-bit pointers, and we can even foresee a point in the not-so-distant future where even these pointers will be insufficient, while the multilevel approach can be implemented using 8 or 16 bit indices at the last level regardless of the size of the reachability set, as long as $\mathcal{R}^{K-1}$ is not too large.

Just as with a single search tree, we can implement our multilevel scheme using splay or AVL trees. However, the choice between storing $\mathcal{R}$ and $\mathcal{U}$ separately or as a single set has subtler implications. If we choose to store them as a single set, distinguishing between explored and unexplored states or, more precisely, being able to access the unexplored states only, still requires one of the approaches discussed for the single-level case (Fig. 4). Since only local states are stored for each level, without repetition, we can no longer use the state array approach of Fig. 4(c). The other two methods, which involve maintaining a list of unexplored states, would only allow us to access nodes in trees at level $K-1$, which are meaningless if we do not know the entire path from level 0. In other words, we would only know the last component of each unexplored state, but not the first $K-1$ components. To obtain all the components, we must add a backward pointer from each node (conceptually, from each tree, since all the nodes in a given tree have the same previous components) to a node in the previous level, and so on. This additional memory overhead is substantial.

With the single-level approach, then, we store $\mathcal{R} \cup \mathcal{U}$ in a single tree, using the state array described in Fig. 4(c). With the multilevel approach, instead, we store $\mathcal{R}$ and $\mathcal{U}$ separately. In this second case, states are truly deleted from $\mathcal{U}$. However, we are free to remove states in any order we choose. When using AVL trees, we use the balance information to choose, at each level, a state in a leaf of the tree in such a way that no rebalancing is ever required.

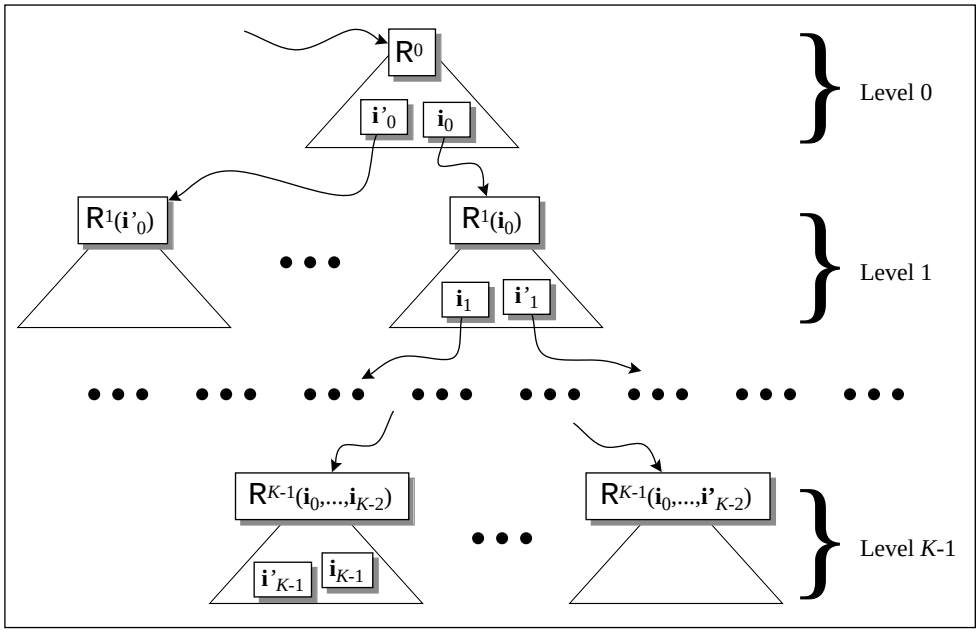

Figure 7: A multilevel storage scheme for $\mathcal{R}$.

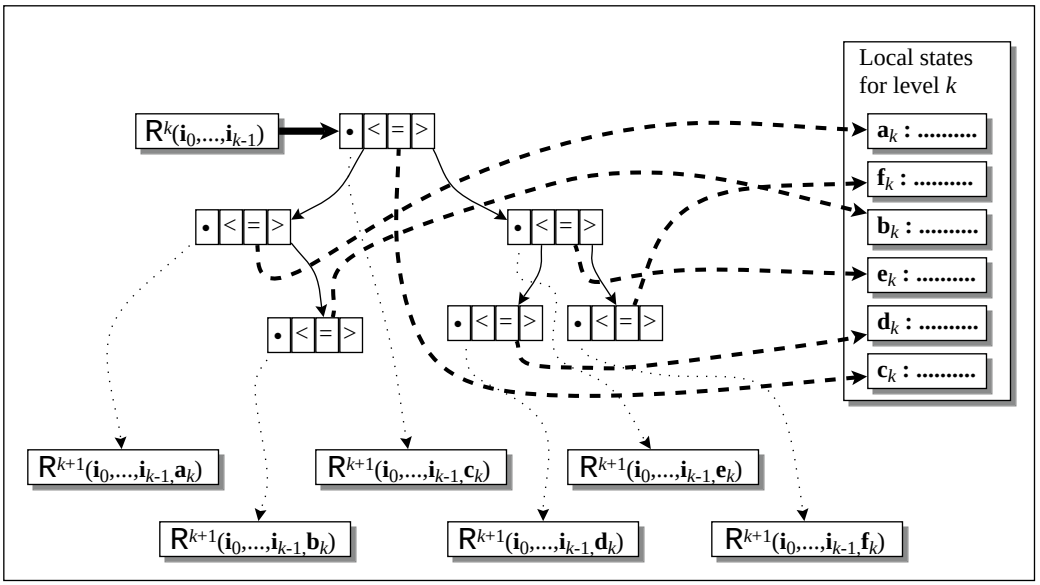

Figure 8: A tree at level $k < K-1$, pointing to trees at level $k+1$.

The total number of nodes used for trees at level k equals the number of reachable substates up to that level, $|\mathcal{R}_0^k|$, hence the total number of tree nodes for our multilevel data structure is

$$\sum_{k=0}^{K-1} |\mathcal{R}_0^k| > |\mathcal{R}_0^{K-1}| = |\mathcal{R}|$$

where the last expression is the number of nodes required in the standard single-level approach of Fig. 4. Two observations are in order:

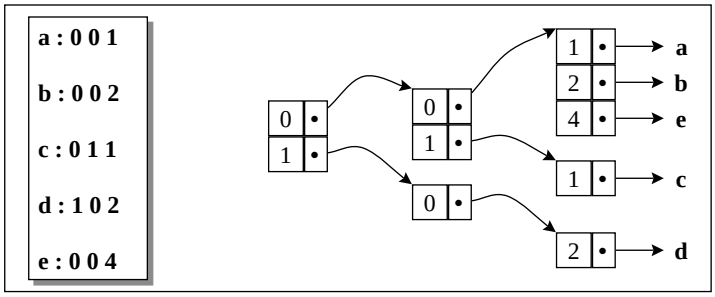

Figure 6: Chiola's storage scheme to store the reachability set of a SPN.

fixed-size vector $\mathbf{i} = (\mathbf{i}_0, \mathbf{i}_1, \ldots, \mathbf{i}_{K-1})$, he proposed the strategy illustrated in Fig. 6, copied from [4]. In the figure, it is assumed that the SPN has three places, and that there are five reachable markings, **a** through **e**. A three-level tree is then used. The first level discriminates markings on their first component (either 0 or 1); the second level discriminates on their second component (also either 0 or 1) within a given first component; finally, the third level fully determines the marking using the third component (1, 2, or 4).

While it is sometimes possible to apply the Kronecker approach where "submodels" are individual places of the SPN [9], this is seldom desirable. Hence, we will use a multilevel approach with K levels, one for each submodel (a sub-GSPN), as shown in Fig. 7. Before explaining this data structure, we need to define the following sets:

- The set of reachable substates up to k:

  $$\mathcal{R}_0^k = \{(\mathbf{i}_0, \ldots, \mathbf{i}_k) \in \mathcal{R}^0 \times \cdots \times \mathcal{R}^k : \exists (\mathbf{i}_{k+1}, \ldots, \mathbf{i}_{K-1}) \in \mathcal{R}^{k+1} \times \cdots \times \mathcal{R}^{K-1}, (\mathbf{i}_0, \ldots \mathbf{i}_{K-1}) \in \mathcal{R}\}.$$

  Clearly, $\mathcal{R}_0^{K-1}$ coincides with $\mathcal{R}$ itself, while $\mathcal{R}_0^0$ coincides with $\mathcal{R}^0$ iff every element of the local state space 0 can actually occur in some global state, and so on.

- The set of reachable local states in $\mathcal{R}^k$ conditioned on a reachable substate $(\mathbf{i}_0, \ldots, \mathbf{i}_{k-1}) \in \mathcal{R}_0^{k-1}$:

  $$\mathcal{R}^k(\mathbf{i}_0, \ldots, \mathbf{i}_{k-1}) = \{\mathbf{i}_k \in \mathcal{R}^k : (\mathbf{i}_0, \ldots, \mathbf{i}_k) \in \mathcal{R}_0^k\}.$$

  Clearly, this set, when defined (i.e., when indeed $(\mathbf{i}_0, \ldots, \mathbf{i}_{k-1}) \in \mathcal{R}_0^{k-1}$), is never empty.

In Fig. 7, level $k \in \{0, \ldots, K-1\}$ contains $|\mathcal{R}_0^{k-1}|$ search trees, each one storing $\mathcal{R}^k(\mathbf{i}_0, \ldots, \mathbf{i}_{k-1})$, for a different reachable substate $(\mathbf{i}_0, \ldots, \mathbf{i}_{k-1}) \in \mathcal{R}_0^{k-1}$. Thus, a generic tree at level $k < K-1$ has the structure shown in Fig. 8. In each one of its nodes, the "•" pointer points to a tree at level $k+1$. In the drawing, the local states for level k are stored in an unsorted dynamically extensible array, in the order in which they are found, and pointed by the "=" pointer in each node. Alternatively, it is possible to store a local state directly in the node. If the submodel is quite complex, this second alternative might be not as memory effective, since a local state can require several bytes for its encoding, which are then repeated in each node, instead of just a pointer. Furthermore, if we can assume an upper bound on the number of local states for a given submodel (e.g., 2^{16}), we can store an index (16 bits) into the local state array, instead of a pointer (usually 32 bits).

```
VectMatrMultiply(in: x, K, R, A^0, A^1, ..., A^{K-1}; inout: y)
 1. for each i ∈ R do
 2.     I ← Ψ(i);
 3.     for each j_0 s.t. A^0_{i_0,j_0} > 0 do
 4.         a_0 ← A^0_{i_0,j_0};
 5.         for each j_1 s.t. A^1_{i_1,j_1} > 0 do
 6.             a_1 ← a_0 · A^1_{i_1,j_1};
    ...
 7.             for each j_{K-1} s.t. A^{K-1}_{i_{K-1},j_{K-1}} > 0 do
 8.                 a_{K-1} ← a_{K-2} · A^{K-1}_{i_{K-1},j_{K-1}};
 9.                 J ← Ψ(j);
10.                 if J ≠ null then
11.                     y_J ← y_J + x_I · a_{K-1};
```

Figure 5: Procedure to multiply a vector by a submatrix of a Kronecker product.

approach based on Kronecker algebra over data structures of size $O(|\mathcal{R}|)$. For example, with the Power method, $\mathbf{x}$ is the old iterate for $\boldsymbol{\pi}$ while $\mathbf{y}$ accumulates what will be the new iterate. We stress that the procedure *VectMatrMultiply* relies heavily on the extreme sparsity of the matrices involved (assumed to be stored in sparse row-wise format), while algorithms such as those proposed by Plateau and Stewart [16] are more appropriate for full matrices.

In the case of superposed GSPNs [12], the test $J \neq$ null of statement 10 is always satisfied, hence, it should be apparent that procedure *VectMatrMultiply* considers all and only the nonzero entries in the submatrix $\mathbf{A}_{\mathcal{R},\mathcal{R}}$ of $\mathbf{A}$. The overall complexity is then affected by the $\log|\mathcal{R}|$ overhead to compute the index J of $\mathbf{j}$ in $\mathcal{R}$, in statement 9, within the innermost for-loop:

$$O\left(\eta(\mathbf{A}_{\mathcal{R},\mathcal{R}}) \cdot \log|\mathcal{R}|\right).$$

Hence, two main limitations in [13] prevent the application of this approach to truly enormous problems. First, the storage of the state space $\mathcal{R}$ and of the iteration vectors, also of size $|\mathcal{R}|$ might require excessive memory. Second, from an execution time standpoint, the $\log|\mathcal{R}|$ factor represents an additional complexity with respect to the traditional approach where the iteration matrix is stored explicitly.

In the next sections, we show how the memory requirements can be reduced, while improving the execution complexity at the same time.

5 A multilevel search tree to store $\mathcal{R}$

We extend work by Chiola, who defined a multilevel technique to store the reachable markings of a SPN [4]. Since the state, or marking, of a SPN with K places can be represented as a

of customers in queue k. However, a realistic model can easily have dozens of queues; so it might be more efficient to partition the queues into K sets, and let $\mathbf{i}_k$ be the number of possible combinations of customers into the queues of partition k. An analogous discussion applies to stochastic Petri nets (SPNs), and even to more complex models, as long as they have a finite state space.

Such a structured model definition is essential to the Kronecker approach for the specification and solution of complex Markov models recently proposed by various authors [2, 3, 12, 13, 15]. The main idea is that the infinitesimal generator $\mathbf{Q}$ of a large model composed of K submodels can be described as (a submatrix of) a sum of Kronecker products of K smaller matrices, each related to a different submodel. Without going into further details, we simply list a few key features of this approach, which prompted much of our work.

- The potential set $\hat{\mathcal{R}}$ is the Cartesian product of K local state spaces, one for each submodel. Our lexical order can be applied just as well to $\hat{\mathcal{R}}$, hence we can define a bijection $\hat{\Psi} : \hat{\mathcal{R}} \to \{0, \ldots, |\hat{\mathcal{R}}| - 1\}$, analogous to Ψ. Indeed, $\hat{\Psi}$ has a fundamental advantage over Ψ: its value equals its argument interpreted as a mixed base integer,

$$\hat{\Psi}(\mathbf{i}) = (\ldots((\mathbf{i}_0)n_1 + \mathbf{i}_1)n_2 \cdots)n_{K-1} + \mathbf{i}_{K-1} = \sum_{k=0}^{K-1} \mathbf{i}_k \cdot n_{k+1}^{K-1},$$

  where $n_i^j = \prod_{k=i}^{j} n_k$.

- The reachability set $\mathcal{R}$ may coincide with $\hat{\mathcal{R}}$, but it is more likely to be much smaller, since many combinations of local states might not occur.

- Numerical solutions compute the steady-state probabilities of each state using either a vector $\hat{\boldsymbol{\pi}}$ of size $|\hat{\mathcal{R}}|$ or a vector $\boldsymbol{\pi}$ of size $|\mathcal{R}|$. In the former case, the probability of state $\mathbf{i} \in \mathcal{R}$ is stored in $\hat{\boldsymbol{\pi}}_{\hat{\Psi}(\mathbf{i})}$, while the entries of $\hat{\boldsymbol{\pi}}$ corresponding to unreachable states are not used, they are initially set to zero and never modified [12]. In the latter case, the same probability is stored in $\boldsymbol{\pi}_{\Psi(\mathbf{i})}$ [13]. We prefer this second approach because it requires potentially much less memory. However, as described by Kemper, its complexity has an additional $O(\log |\mathcal{R}|)$ multiplicative factor, since, for each reachable state $\mathbf{i} \in \mathcal{R}$ and for each possible transition from $\mathbf{i}$ to $\mathbf{j}$, the index $\Psi(\mathbf{j})$ of $\mathbf{j}$ in $\boldsymbol{\pi}$ must be computed using a binary search.

- Any steady-state or transient solution method that avoids storage and execution complexity of order $|\hat{\mathcal{R}}|$ must iterate over the elements of $\mathcal{R}$ in some order. The lexical order is acceptable, especially for methods such as Power, Jacobi, or Uniformization, which are not affected by the order in which the variables are considered. To clarify the relevant complexity issues, we show procedure *VectMatrMultiply* in Fig. 5, to perform the assignment

$$\mathbf{y} \leftarrow \mathbf{y} + \mathbf{x} \cdot \left(\mathbf{A}^0 \otimes \mathbf{A}^1 \otimes \cdots \otimes \mathbf{A}^{K-1}\right)_{\mathcal{R},\mathcal{R}},$$

  where $\mathbf{A}^k$ is a matrix of order n_k and the subscript "$\mathcal{R}, \mathcal{R}$" indicates the submatrix of the Kronecker product corresponding to the reachable states only. This is the main operation needed when solving for the steady state or transient probabilities for any

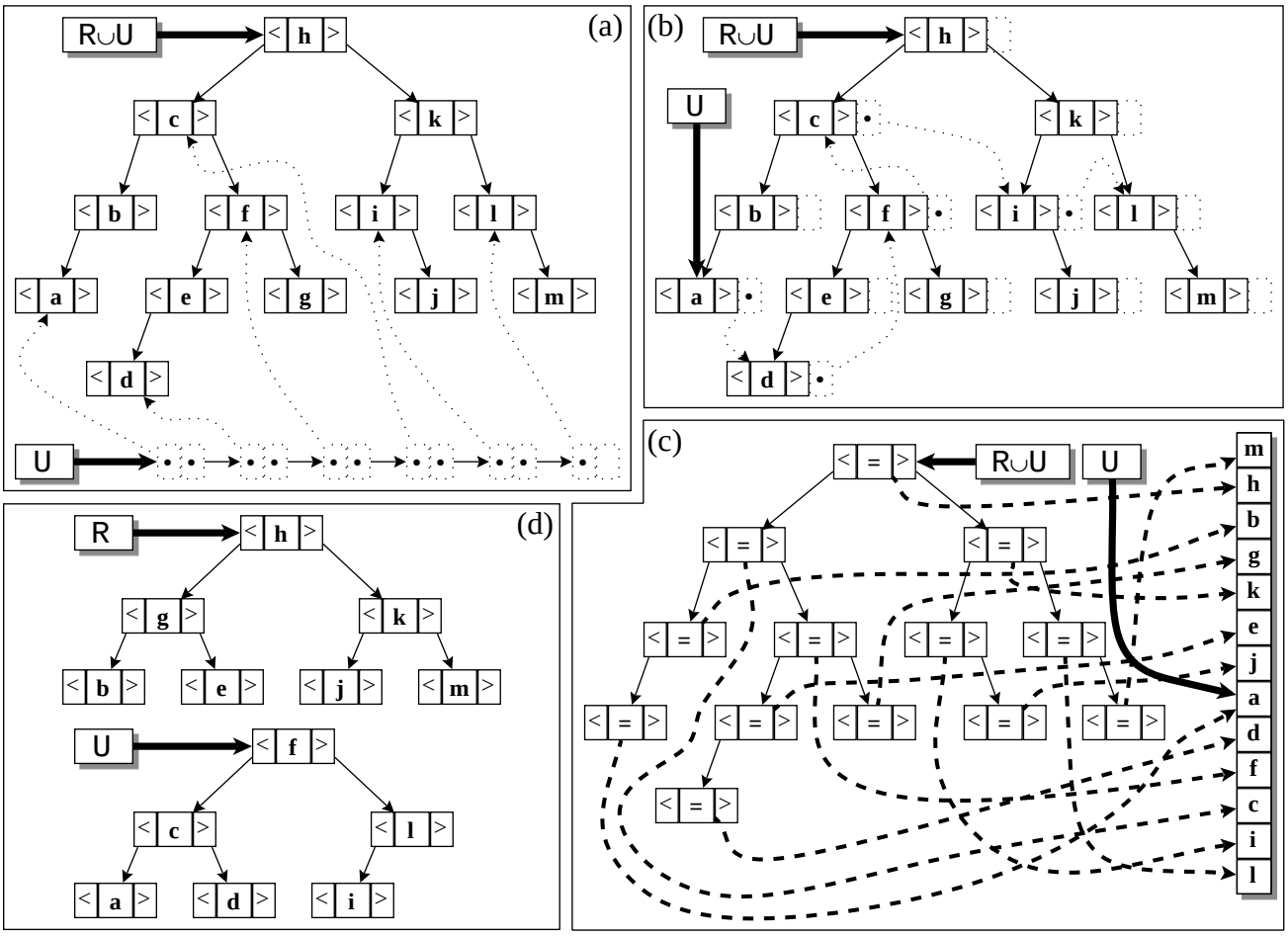

Figure 4: Four simple storage schemes for $\mathcal{R}$ and $\mathcal{U}$.

N	$\lvert\mathcal{R}\rvert$ for kanban-timed	$\lvert\mathcal{R}\rvert$ for kanban-immediate	$\lvert\mathcal{R}\rvert$ for FMS
1	160	152	54
2	4,600	3,816	810
3	58,400	41,000	6,520
4	454,475	268,475	35,910
5	2,546,432	1,270,962	152,712
6	11,261,376	4,785,536	537,768
7	-	-	1,639,440

Table 1: Reachability set size for the three models, as a function of N.

4 Structured state spaces

We now assume that $\hat{\mathcal{R}}$ can be expressed as the Cartesian product of K "local" state spaces: $\hat{\mathcal{R}} = \mathcal{R}^0 \times \cdots \times \mathcal{R}^{K-1}$, and that $\mathcal{R}^k$ is simply $\{0, \ldots n_k - 1\}$, although n_k might not be known in advance. Hence, a (global) state is a vector $\mathbf{i} \in \{0, \ldots n_0 - 1\} \times \cdots \times \{0, \ldots n_{K-1} - 1\}$. For example, if the high-level model is a single-class queuing network, $\mathbf{i}_k$ could be the number

We consider two alternatives, both based on binary search trees: splay trees [11] and AVL trees [14]. The execution complexity[2] of *BuildRS* when using either method is

$$O\left(|\mathcal{R}| \cdot |\mathcal{E}| \cdot C_{Active} + |\mathcal{A}| \cdot (C_{New} + \log |\mathcal{R}| \cdot C_{Compare})\right),$$

where C_x is the (average) cost to a call to procedure x. For AVL trees, the storage complexity, expressed in bits, is

$$O\left(|\mathcal{R}| \cdot (B_{state} + 2B_{pointer} + 2)\right),$$

where B_{state} is the (average) number of bits to store a state, $B_{pointer}$ is the number of bits for a pointer. The additional two bits are used to store the node balance information. For splay trees, the complexity is the same, except for the two-bit balance, which is not required.

Regarding the use of binary search trees in *BuildRS*, we have another choice to make. We can use a single tree to store $\mathcal{R} \cup \mathcal{U}$ in *BuildRS* as shown in Fig. 4(a,b,c), or two separate trees, as in Fig. 4(d). Using a single tree has some advantages:

- No deletion, possibly requiring a rebalancing, is needed when moving a state from $\mathcal{U}$ to $\mathcal{R}$.
- Procedure *SearchInsert* needs to perform a single search with $\log(|\mathcal{R} \cup \mathcal{U}|)$ comparisons, instead of two searches in the two trees for $\mathcal{R}$ and $\mathcal{U}$, which overall require more comparisons (up to twice as many), since

  $$\log |\mathcal{R}| + \log |\mathcal{U}| = \log(|\mathcal{R}| \cdot |\mathcal{U}|) > \log(|\mathcal{R}| + |\mathcal{U}|) = \log(|\mathcal{R} \cup \mathcal{U}|)$$

  (of course, we mean "the current $\mathcal{R}$ and $\mathcal{U}$" in the above expressions).

However, it also has disadvantages. As explored and unexplored states are stored together in a single tree, there must be a way to identify and access the nodes of $\mathcal{U}$. We can accomplish this by:

- Maintaining a separate linked list pointing to the unexplored states. This requires an additional $O(2|\mathcal{U}| \cdot B_{pointer})$ bits, as shown in Fig. 4(a).
- Storing an additional pointer in each tree node, pointing to the next unexplored state. This requires an additional $O(|\mathcal{R}| \cdot B_{pointer})$ bits, as shown in Fig. 4(b).
- Storing the markings in a dynamic array structure. In this case, the nodes of the tree contain indices to the array of markings, instead of the markings themselves. If markings are added to the array in the order they are discovered, $\mathcal{R}$ occupies the beginning of the array, while $\mathcal{U}$ occupies the end. This requires an additional $O(|\mathcal{R}| \cdot B_{index})$ bits, where B_{index} is the number of bits required to index the array of markings, as shown in Fig. 4(c)

The size of $\mathcal{R}$ for our three models is shown in Table 1, as a function of N. Since the models are GSPNs, they give rise to "vanishing markings", that is, markings enabling only immediate transitions, in which the sojourn time is zero. In all our experiments, these markings are eliminated "on the fly" [8], so that all our figures regarding $\mathcal{R}$ reflect only the "tangible" reachable markings.

[2] We leave known constants inside the big-O notation to stress that they do make a difference in practice, even if they are formally redundant.

```
BuildRS(in: Events, Active, New; out: R);
  1. R ← ∅; /* R: states explored so far */
  2. U ← {i^0}; /* U: states found but not yet explored */
  3. while U ≠ ∅ do
  4.       i ← ChooseRemove(U);
  5.       R ← R ∪ {i};
  6.       for each e ∈ E s.t. Active(e, i) = True do
  7.             j ← New(e, i);
  8.             SearchInsert(j, R ∪ U, U);
```

Figure 3: Procedure *BuildRS*

events. Formally, $\mathcal{R}$ is the smallest subset of $\hat{\mathcal{R}}$ satisfying:

- $\mathbf{i}^0 \in \mathcal{R}$, and
- $\mathbf{i} \in \mathcal{R} \wedge \exists e \in \mathcal{E}, Active(e, \mathbf{i}) = True \wedge \mathbf{i}' = New(e, \mathbf{i}) \Rightarrow \mathbf{i}' \in \mathcal{R}$.

While a high-level model usually specifies other information as well (the stochastic timing of the events, the measures that should be computed when solving the model, etc.), the above description is sufficient for now, and it is not tied to any particular formalism.

$\mathcal{R}$ can be generated using the state-space exploration procedure *BuildRS* shown in Fig. 3, which terminates if $\mathcal{R}$ is finite. This is a search of the graph implicitly defined by the model. Function *ChooseRemove* chooses an element from its argument (a set of states), removes it, and returns it. Procedure *SearchInsert* searches the first argument (a state), in the second argument (a set of states) and, if not found, it inserts the state in its third argument (also a set of states).

The "reachability graph", $(\mathcal{R}, \mathcal{A})$, has as nodes the reachable states, and an arc from $\mathbf{i}$ to $\mathbf{j}$ iff there is an event e such that $Active(e, \mathbf{i}) = True$ and $New(e, \mathbf{i}) = \mathbf{j}$. Depending on the type of analysis, the arc might be labelled with e; this might result in a multigraph, if multiple events can cause the same change of state. While we focus on $\mathcal{R}$ alone, the size of $\mathcal{A}$ affects the complexity of *BuildRS*.

A total order can be defined over the elements of $\mathcal{R}$: $\mathbf{i} < \mathbf{j}$ iff $\mathbf{i}$ precedes $\mathbf{j}$ in lexical order. We can then define a function $Compare : (\hat{\mathcal{R}} \times \hat{\mathcal{R}}) \rightarrow \{-1, 0, 1\}$ returning -1 if the first argument precedes the second one, 1 if it follows it, and 0 if the two arguments are identical. This order allows the efficient search and insertion of a state during *BuildRS*. Once $\mathcal{R}$ has been built, we can define a bijection $\Psi : \mathcal{R} \rightarrow \{0, \ldots, |\mathcal{R}| - 1\}$, such that $\Psi(\mathbf{i}) < \Psi(\mathbf{j})$ iff $\mathbf{i} < \mathbf{j}$.

Common techniques to store and search the sets $\mathcal{R}$ and $\mathcal{U}$ include hashing and search trees. Hashing would work reasonably well if we had a good bound on the final size of the reachability set $\mathcal{R}$, but this is not usually the case. We prefer search trees: when kept balanced, they have more predictable behavior than hashing.

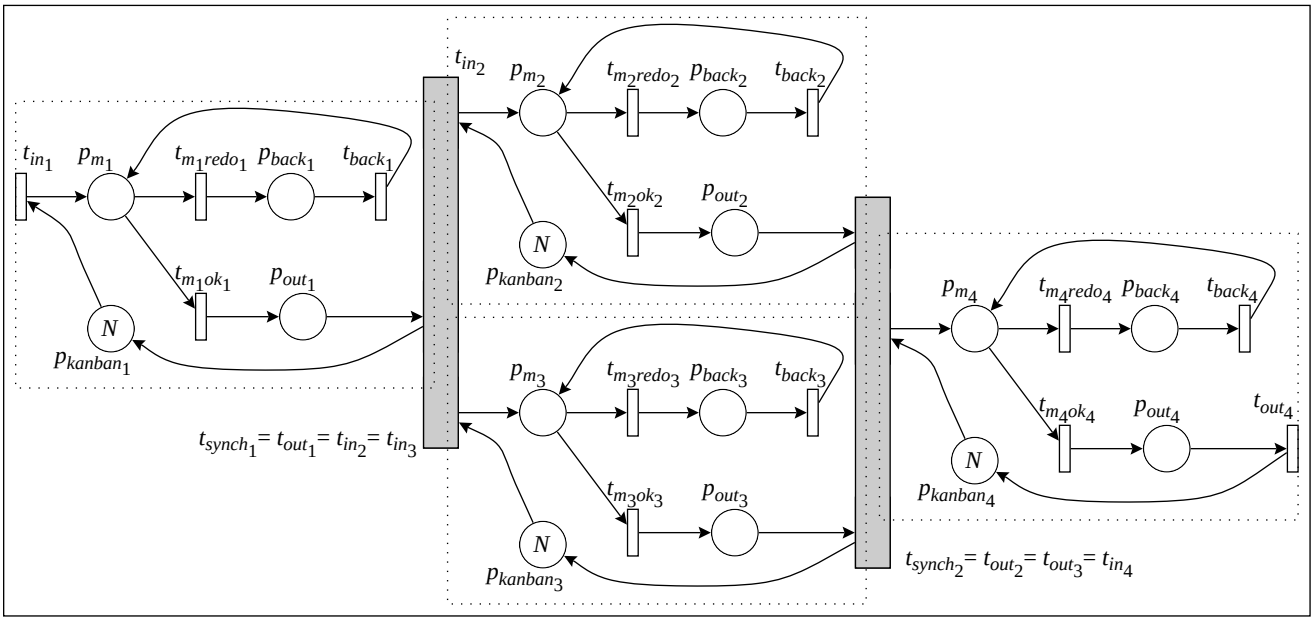

Figure 1: The GSPN of a Kanban system.

3 The reachability set

From the high-level description, we can build the "reachability set", $\mathcal{R} \subseteq \hat{\mathcal{R}}$. This is the set of states that can be reached from $\mathbf{i}^0$ through the occurrence of any sequence of enabled

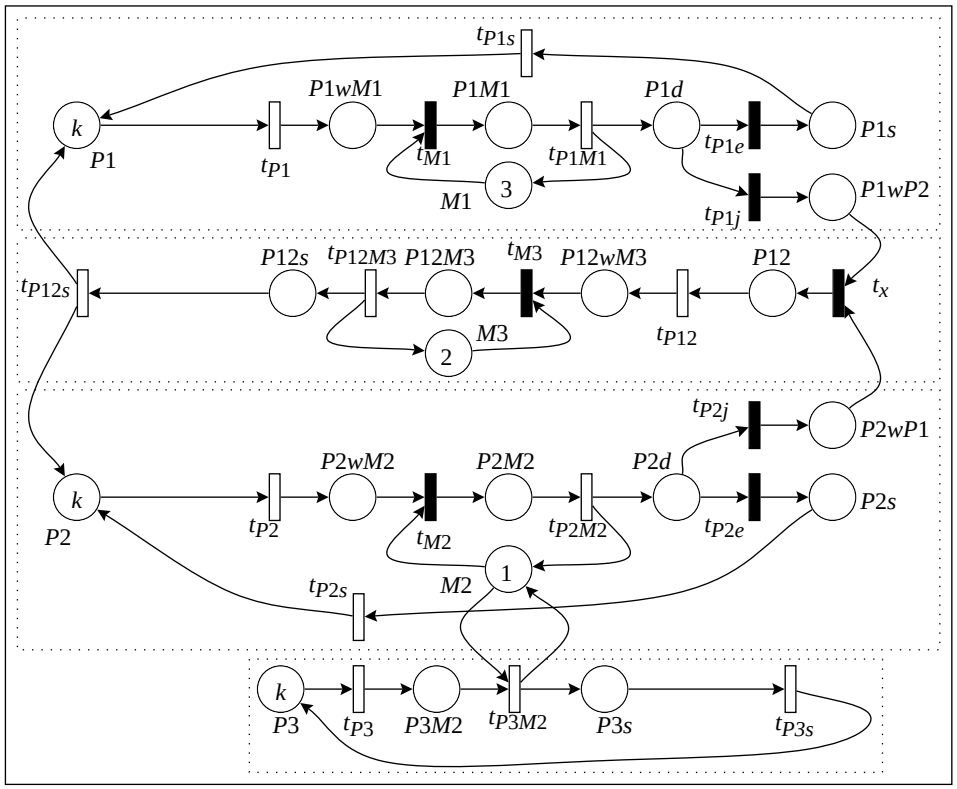

Figure 2: The GSPN of a FMS system.

this case, the storage bottleneck is indeed due to the state space and the probability vectors allocated when computing the numerical solution.

Sections 2, 3, and 4 define the type of high-level formalisms we use, discuss the reachability set and its storage, and motivate the decomposition of a model into submodels, respectively. Our main contributions are in Sections 5 and 6, where we introduce a multilevel data structure and show how it can be used to save both memory and execution time when building the reachability set. Section 7 further explores storing and searching the reachability set after it has been built. Finally, Section 8 contains our final remarks.

2 High-level model description

We implemented our techniques in SMART [7], using the GSPN formalism [1, 5]. However, these techniques can be applied to any model expressed in a "high-level formalism" which defines:

- The "potential set", $\hat{\mathcal{R}}$. This is a discrete set, assumed finite, to which the states of the model must belong[1].
- A finite set of possible events, $\mathcal{E}$.
- An initial state, $\mathbf{i}^0 \in \hat{\mathcal{R}}$.
- A boolean function defining whether an event is active (can occur) in a state, $Active : \mathcal{E} \times \hat{\mathcal{R}} \rightarrow \{True, False\}$.
- A function defining the effect of the occurrence of an active event on a state: $New : \mathcal{E} \times \hat{\mathcal{R}} \rightarrow \hat{\mathcal{R}}$.

Fig. 1 and 2 show the two GSPNs used throughout this paper to illustrate the effect of our techniques (ignore for the moment the dashed boxes). The first GSPN models a kanban systems, from [9, 17]. It is composed of four instances of essentially the same sub-GSPN. The synchronizing transitions t_{synch_1} and t_{synch_2} can be either both timed or both immediate, we indicate the two resulting models as kanban-timed and kanban-immediate. The second GSPN models a flexible manufacturing system, from [10], except that the cardinality of all arcs is constant, unlike the original model (this does not affect the number of reachable markings). We indicate this model as FMS. We do not describe these models in more detail, the interested reader is referred to the original publications where they appeared.

[1] A point about notation: we denote sets by upper case calligraphic letters. Lower and upper case bold letters denote vector and matrices, respectively. $\eta(\mathbf{A})$ is the number of nonzero entries in a matrix $\mathbf{A}$; $\mathbf{A}_{i,j}$ is the entry in row i and column j of $\mathbf{A}$; $\mathbf{A}_{\mathcal{X},\mathcal{Y}}$ is the submatrix of $\mathbf{A}$ corresponding to the set of rows $\mathcal{X}$ and the set of columns $\mathcal{Y}$.

1 Introduction

Extremely complex systems are increasingly common. Various types of high-level models are used to describe them, study them, and forecast the effect of possible modifications. Unfortunately, the logical and dynamic analysis of these models is often hampered by the combinatorial explosion of their state spaces, a problem inherent with discrete-state systems. Possible solutions are:

- Use approximate or bounding analysis methods. These are particularly effective when studying the performance or reliability of a system described by a stochastic model (such as a queueing network or a stochastic Petri net).

- Use discrete-event simulation. This does not require the generation and storing of the entire state space. However, it is a Montecarlo method, and hence it can only state that a certain condition is (or is not) encountered with a certain probability. For example, if the simulation runs performed during an experiment don't find a deadlock, we cannot conclude with certainty that the system is deadlock-free.

- Use exact methods that cope with large state spaces. These include the use of better algorithms and data-structures and of distributed approaches exploiting multiple workstations, plus, of course, the trivial (but expensive) approach of increasing the amount of RAM.

We stress that the above approaches are not mutually exclusive. Indeed, it is quite common to use exact methods in the early stages of a system's design, when studying rough models over a wide range of parameter combinations, and then focus on more detailed models using approximate methods or simulation. Indeed, many approximate decomposition approaches use exact methods at the submodel level, thus introducing errors only when exchanging or combining (exact) results from different submodels. Then, the ability of applying exact methods to larger models will normally result in better approximations: it is likely that the study of a large model decomposed into four medium-size submodels, each solved with exact methods, will be more accurate than that of the same model decomposed into ten small-size submodels.

The first problem when applying exact methods is the generation and storage of the state space. For performance or reliability analysis, the model is then used to generate a stochastic process, often a continuous-time Markov chain (CTMC), which is solved numerically for steady-state or transient measures. The state space itself can be of interest since it can be used to answer questions such as existence of deadlocks and livelocks or liveness.

In this paper, we focus on techniques to store and search the state space. However, while we do not discuss the solution of the stochastic process explicitly, our results apply not only to the "logical" analysis of the model, but even more so to its "stochastic analysis" (indeed, this was our initial motivation). This is particularly true given the recent developments on the solution of complex Markov models using Kronecker operators [2, 3, 9, 12, 13, 15, 17] which do not require the storage of the infinitesimal generator of the CTMC explicitly. In

Storage alternatives for large structured state spaces*

Gianfranco Ciardo Andrew S. Miner
Dept. of Computer Science
College of William and Mary
Williamsburg, VA 23187-8795, USA
{ciardo,asminer}@cs.wm.edu

Abstract

We consider the problem of storing and searching a large state space obtained from a high-level model such as a queueing network or a Petri net. After reviewing the traditional technique based on a single search tree, we demonstrate how an approach based on multiple levels of search trees offers advantages in both memory and execution complexity, and how solution algorithms based on Kronecker operators greatly benefit from these results. Further execution time improvements are obtained by exploiting the concept of "event locality." We apply our technique to three large parametric models, and give detailed experimental results.

*This research was partially supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-19480 while the first author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681-00091 and by the Center for Advanced Computing and Communication under Contract 96-SC-NSF-1011