

Bottom-Up Beats Top-Down Hands Down*

Mirco Tribastone[†]

Abstract

In PEPA, the calculation of the transitions enabled by a process accounts for a large part of the time for the state space exploration of the underlying Markov chain. Unlike other approaches based on recursion, we present a new technique that is iterative—it traverses the process’ binary tree from the sequential components at the leaves up to the root. Empirical results shows that this algorithm is faster than a similar implementation employing recursion in Java. Finally, a study on the user-perceived performance compares our algorithm with those of other existing tools (ipc/Hydra and the PEPA Workbench).

1 Introduction

PEPA [1] is a stochastic process algebra that permits performance evaluation through the analysis of a Continuous-Time Markov Chain underlying a model description. Here follows a common sequence of tasks to obtain performance measures: 1) Model development; 2) Static analysis; 3) State space exploration; 4) Solution of the Markov chain; 5) Throughput, utilisation, passage time analysis. This process is typically assisted by software tools. PEPA is supported by ipc/Hydra [2] for passage-time analysis and by the PEPA Plug-in Project [3] and the PEPA Workbench [4] for Markovian steady-state analysis.

As far as computational cost is concerned, the model development stage can be characterised by the modeller’s typing rate, which is generally many orders of magnitude slower than current computer architectures. Static analysis is designed for the specific task of detecting potential problems by running a battery of lightweight utilities on the model’s abstract syntax tree. As a result, such tools are so fast that in current implementations they are scheduled every time the input file is changed. On the other hand, the expressiveness of process algebras is such that even very compact descriptions of the system can lead to very large state spaces, the exploration of which results in a long-running process, since the problem is often exponential with the number of states. Numerical solution of Markov chains is also a computationally expensive task, and many methods and analytical results are available in the research literature [5]. Throughput, utilisation and passage time analysis are less problematic as they usually involve $O(n)$ operations when provided with the stationary distribution.

*Stephen Gilmore’s suggestion for this title is gratefully acknowledged.

[†]LFCS, The University of Edinburgh, Scotland, UK. Email: mtribast@inf.ed.ac.uk

In this paper we deal with the exploration of the state space of PEPA models. It is important to achieve this task effectively for a number of reasons. As well as being an intermediate result for the performance evaluation problem, the state space of a system is important *per se* because it represents the end product for other kind of analyses such as deadlock detection and liveness analysis. In addition, it is a useful debugging tool for the modeller to figure out, for example, whether a particular sequence of states can be observed or if a given state enables an action. The PEPA Plug-in project, for instance, offers tools which guide the user through these tasks. For a user-interface designer point of view, it is important that the state space is built promptly in order for the application to be as responsive as possible.

A common approach to state space generation is depth first search. This makes use of two data structures: a stack of unexplored states and a table of explored states (the state space). The algorithm is initialised by adding the initial state of the system to the table and the stack. For each state s popped off the stack, a function `successor(s)` calculates its transitions. Then, if the target of a transition is not contained in the table, it is added to the state space and pushed onto the stack. The algorithm terminates when the stack is empty.

We identify two distinct performance issues with this algorithm. The first is concerned with a space/time trade-off for the layout and management of the table of the explored states. Here we want to be able to successfully explore a large state space while keeping the memory footprint low and delivering fast look-up. Several techniques tackling this problem are available in the research literature [6]. The second issue—which is the focus of this paper—is about optimising the execution of the `successor` function. Later we will present empirical data that shows that this amounts to a significant percentage of the overall execution time of the state space exploration.

In the remainder of this paper we review the state space exploration techniques employed by the PEPA Workbench and the current version of PEPAto, which both involve a recursive procedure which we call the *top-down algorithm*. Then we will present a new approach that exploits some features of the structure of PEPA descriptions to develop an iterative technique which we call the *bottom-up algorithm*. We also discuss the implementation of this algorithm in Java and an empirical study that shows how the bottom-up algorithm outperforms the top-down one. Finally, we present a study on the user-perceived performance of the state space exploration in PEPA by comparing our algorithm with those of ipc/Hydra and the PEPA Workbench.

2 The Top-Down State Space Builder

The operational semantics of PEPA is the basis for the exploration of the state space. When a system is in a given state, a set of rules can be applied to determine the *activity multiset*, i.e. the multiset of all the enabled transitions from that state. Due to lack of space, we limit ourselves to a more detailed description of the cooperation operator, whose operational semantics is as follows:

$$\frac{E \xrightarrow{(\alpha, r_1)} E' \quad F \xrightarrow{(\alpha, r_2)} F'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha, R)} E' \underset{L}{\bowtie} F'} \quad R = \frac{r_1}{r_\alpha(E)} \frac{r_2}{r_\alpha(F)} \min \{r_\alpha(E), r_\alpha(F)\}$$

Listing 1: Top-Down Approach

```

public ActivityMultiset create(Cooperation coop) {
    ActivityMultiset left = create(coop.getLeft());
    ActivityMultiset right = create(coop.getRight());
    ActivityMultiset result = new ActivityMultiset();
    for (Transition leftTrst : left) {
        if ((coop.getActionSet().contains(leftTrst.getAction())) {
            for (Transition rightTrst : right) {
                if (rightTrst.getAction()
                    .equals(leftTrst.getAction())) {
                    /* creates shared transition */
                    Cooperation resCoop = new Cooperation();
                    resCoop.setLeft(leftTrst.getTarget());
                    resCoop.setRight(rightTrst.getTarget());
                    Rate rate = createSharedRate(...);
                    Transition tr = new Transition(
                        leftTrst.getAction(), rate, resCoop);
                    result.add(tr);
                }
            }
        }
    }
}

```

The rule states that in order for a cooperation to enable an activity α both sides must have α in their activity multisets. If this condition is satisfied, a shared action is enabled at a rate R which depends on the individual rates of the cooperating processes.

We note that this definition is intrinsically recursive. Hence, an implementation with programming languages with efficient support for recursion is straightforward. In Listing 1 we show a code snippet of this implementation for the calculation of shared actions of a cooperation. We assume that a `create` function is available for every operator of PEPA. Thus, the `successor(s)` function is simply the invocation of `create(s)`, which returns a multiset of enabled transitions. The first two lines of the method are where recursion occurs. The *exit condition* of such a recursion is represented by the invocation of `create(Prefix)`, which simply returns itself. We refer to this approach as the *top-down algorithm* because it traverses the state's binary tree from the root down to the leaves—the system's sequential components. Variants of this algorithm are employed by both the PEPA Workbench and PEPAto.

We observe that in the worst-case—a system of all left-associated co-operations of sequential components—the depth of this recursion is the number of the sequential components. In the next section we present an iterative approach which aims to reduce such stack frame allocation overhead.

3 The Bottom-Up Approach

We consider the class of PEPA models described by the following two-level grammar:

$$\begin{aligned} S &::= (\alpha, r).S \mid S + S \mid A \\ P &::= S \mid P \underset{L}{\boxtimes} P \mid P/L \end{aligned}$$

This class has the property that the configuration of the system is static, i.e., the way the sequential components are composed together does not change over the state space. Hence, we only need to traverse the tree of the initial state to obtain the structure of every state. Also, such a description leads to a finite-state Markov chain so we need not be concerned about infinite-state solution methods here.

A depth-first visit of the tree allows us to initialise two data structures: the *state description vector* and the model's *composition tree*.

The state description vector is a flattened representation of a state as an array of component identifiers. The length of the array is the number of sequential components in the system equation. The identifiers are added into the array in the same order as they are visited during the traversal of the tree.

The composition tree is based on two types. A **Component** offers a method to retrieve its first-step derivatives (`getDerivatives`) and a method to calculate the apparent rate of an action type (`getApparentRate`). It has three main fields. Two fields hold the position (indicated as `offset` and `length`) of that component in the state description vector. The third is a set of action types (`hidingSet`) which is not empty if a hiding operator is applied to the component. An **Operator** extends **Component** with two fields of type **Component** for its operands, `leftChild` and `rightChild`. It also has a method `compose` responsible for generating the first-step derivatives. This method is similar to the `create(Cooperation)` of the recursive version. The composition tree stores the operators in the reverse order from the order in which they have been visited during initialisation.

In order to show how the composition tree is built, let us consider a simple example of a system of two processes which need exclusive access to a resource in order to carry out some task. This can be modelled in PEPA as follows:

$$\begin{aligned} Process &\stackrel{\text{def}}{=} (use, r_1).Process' \\ Process' &\stackrel{\text{def}}{=} (task, r_2).Process \\ Resource &\stackrel{\text{def}}{=} (use, r_3).Resource' \\ Resource' &\stackrel{\text{def}}{=} (update, r_4).Resource \end{aligned}$$

$$System \stackrel{\text{def}}{=} (Process \parallel Process) \underset{\{use\}}{\boxtimes} Resource$$

The state description vector of the initial state of the system is [**Process**, **Process**, **Resource**]. Figure 1 depicts its binary tree. Each node is labelled with the corresponding PEPA component. In addition, the nodes are annotated with the state of the instances of **Operator** and **Component** resulting from the visit of the tree.

The `successor(s)` algorithm is divided into two parts. An initialisation phase provides each sequential **Component** with the information on apparent rates and the first-step derivatives corresponding to each local state. (Note that such information can be pre-computed during the initialisation stage of

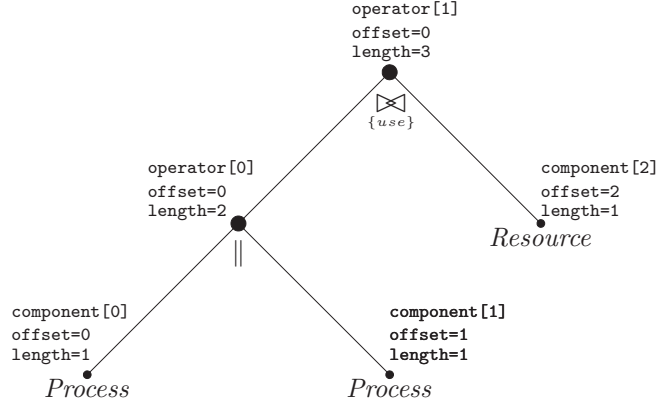


Figure 1: Binary tree of the initial state of the system.

the state space exploration algorithm.). Then, `compose` is sequentially called on each operator. We observe that the first operator in the list has always sequential components as children, for which the calculation of the activity multiset does not involve recursion. After `compose` is called upon it, its first step derivatives and apparent rates are available to its parent, whose `compose` invocation does not require recursion either. Finally, the algorithm returns with the first step derivatives of the last operator. Because this approach traverses the tree from the leaves up to the root, we refer to this as the *bottom-up* algorithm. We observe that the number of sequential components varies the number of iterations in this cycle, but no stack frame allocation for recursion occurs.

4 Evaluation

We provided a Java implementation of the bottom-up algorithm as a module of PEPAto [3]. Our aim is to compare its performance against PEPAto’s top-down implementation of the state space explorer. We equipped both modules with hooks for gathering wall-clock execution times. The accuracy of these data is up to the precision of `java.lang.System.nanoTime()`. To keep this evaluation unbiased, the algorithms share a common set of data structures, such as those for state descriptors, action types, and those supporting depth-first search. Hence, the measures reflect the relative effectiveness of the two algorithms.

We consider the following notation for comparing the execution times of the two algorithms. The overall wall-clock execution time of the state space exploration T is the sum of three terms: T_i , the algorithm set-up time; T_s , the overall time for the calculation of the `successor` function; T_l , the overall time for the management of the state space table. We use the superscripts *bu* and *td* to refer to the bottom-up and the top-down algorithm, respectively. Unless otherwise stated, the results are averaged over 20 independent runs. The platform used to obtain the results presented in this section was: Intel Xeon

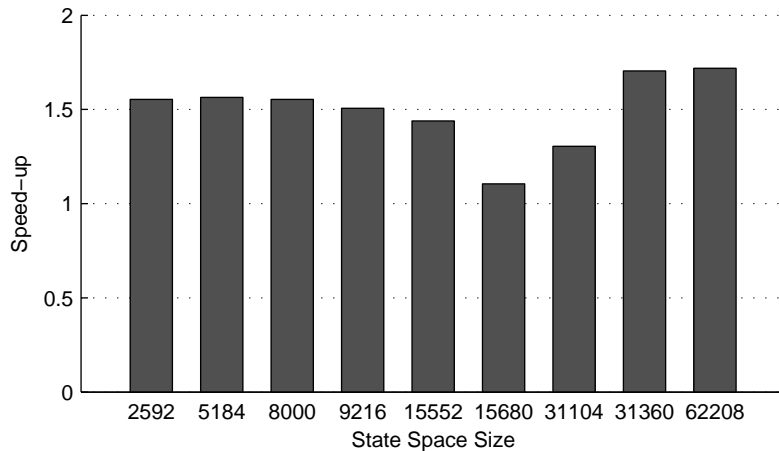


Figure 2: Speed-up of the bottom-up algorithm.

3.20 GHz, 2 GB RAM, Microsoft Windows XP x64 Edition SP2, Sun JVM 1.6 Update 1.

In order to evaluate the performance improvement of the bottom-up algorithm we define the speed-up as:

$$S = \frac{T_i^{td} + T_s^{td}}{T_i^{bu} + T_s^{bu}}$$

Here we include the set-up time, because it is algorithm-dependent. Figure 2 shows speed-up results for a set of PEPA models with state space sizes ranging from 2592 to 62208. We note that the bottom-up algorithm always outperforms the top-down one. The increase depends on the model under study, but no clear dependence on the size of the model has been found. This is the subject of ongoing work.

We carried out code optimisation on the bottom-up algorithm. Changes affected many parts, including the management of the depth-first search structures and the state descriptor. As a result, an unbiased comparison of this version with the recursive algorithm is not possible. In Fig. 3 we present results on the breakdown of the state space exploration time in the same case tests as above run through the optimised version. In all the cases, the evaluation of `successor` accounted for over 80% of the overall time. The algorithm set-up figures are barely visible in the graph, as this phase took up less than 0.7% of the time on average.

4.1 User-perceived Performance

To emphasise how the user-perceived performance is affected by changes in the state space exploration algorithm, we compare overall execution times of three tools supporting PEPA: `ipc/Hydra` (IH), the PEPA Workbench (WB) and the proposed bottom-up implementation (BU) available in the PEPAto library. In this comparison we also include parsing execution times. Such measures were obtained using the Unix `time` utility for IH and `java.lang.System.nanoTime()`

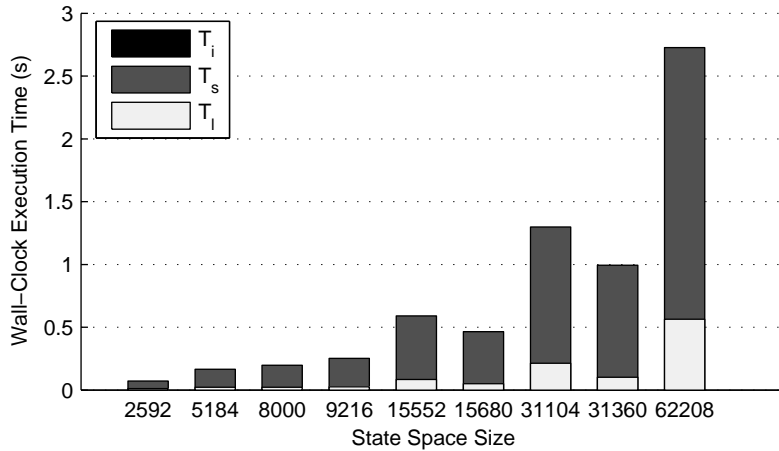


Figure 3: Breakdown of the overall wall-clock execution time in the optimised version of the bottom-up algorithm.

for both BU and WB. Results of overall execution times against models with different state space sizes are plotted in Fig. 4 (the y axis is in log-scale). (These figures do not take into account the time necessary to start the JVM and Cygwin.)

The graph shows that BU performs well in these tests, being two times as fast as IH in the worst case. We also note that WB is at least two orders of magnitude slower than both IH and BU; an `OutOfMemoryError` thrown by WB has prevented us from calculating the execution time in the 62208-state model.

The execution time in both BU and WB increases monotonically with the state space size, whereas a different behaviour is observed in IH. Here, we notice that other factors have an impact on the performance of the algorithm. For instance, the third and the fourth model required the same amount of time for state space generation, despite the latter being twice as large as the former. We believe this because of an efficient way in IH to calculate first-step derivatives of independent copies of sequential components. As a matter of fact, these two models have the same structure, except for two sequential components which are replicated in the system equation of the bigger one.

5 Conclusion

The exploration of the state space of the underlying Markov chain of a PEPA model is a crucial stage of the modelling process, as it enables both qualitative and quantitative analysis. We discussed an iterative algorithm for the efficient calculation of the enabled transitions of a PEPA process. We carried out an experimental study suggesting that the tuning of this algorithm has a profound impact on the user-perceived software performance, as it accounts for over 80% of the overall state space exploration time. Our approach outperforms an equivalent recursive implementation, and a study showed that our approach is faster than the other state space generation tools for PEPA.

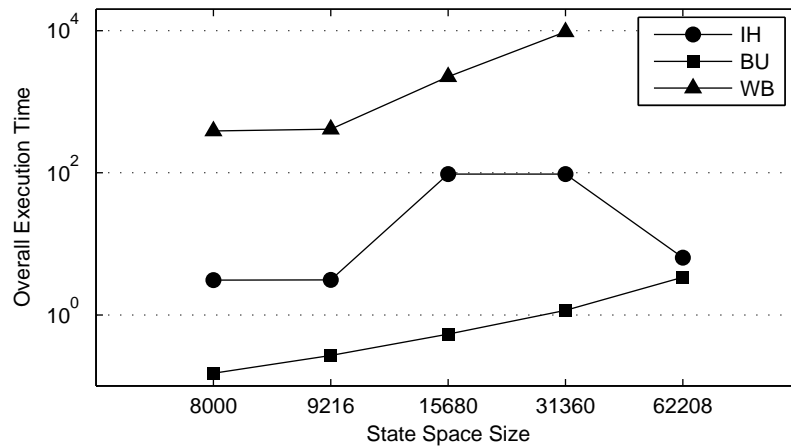


Figure 4: State space exploration in software tools for PEPA.

Acknowledgement

The author would like to thank Stephen Gilmore for very insightful comments on this work.

The author is supported by the project SENSORIA, IST-2005-016004.

References

- [1] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
- [2] J.T. Bradley, N.J. Dingle, S.T. Gilmore, and W.J. Knottenbelt. Derivation of passage-time densities in PEPA models using IPC: The Imperial PEPA Compiler. In *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, October 2003.
- [3] M. Tribastone. The PEPA Plug-in Project. In *Proceedings of the 4th International Conference on the Quantitative Evaluation of Systems (QEST) 2007*, 9 2007.
- [4] S. Gilmore and J. Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, number 794 in Lecture Notes in Computer Science, pages 353–368, Vienna, May 1994. Springer-Verlag.
- [5] W.F. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, Princeton, New Jersey, 1994.
- [6] W.J. Knottenbelt. *Performance Analysis of Large Markov Models*. PhD thesis, Imperial College of Science, Technology and Medicine, 2000.