Pontifícia Universidade Católica do Rio Grande do Sul
Programa de Pós-graduação em Ciência da Computação

THAIS CHRISTINA WEBBER DOS SANTOS

Reducing the Impact of State Space Explosion in Stochastic Automata Networks

Porto Alegre, Brasil
2009

THAIS CHRISTINA WEBBER DOS SANTOS

Reducing the Impact of State Space Explosion in Stochastic Automata Networks

Tese de Doutorado apresentada como requisito para obtenção do título de Doutor em Ciência da Computação pelo Programa de Pós-graduação da Faculdade de Informática. Área de concentração: Ciência da Computação.

Orientador: Paulo Henrique Lemelle Fernandes
Co-orientador: Jean-Marc Vincent

Porto Alegre, Brasil
2009

Abstract

The solution of Markovian models with large state spaces is one of the major challenges in performance evaluation. Structured formalisms such as Stochastic Automata Networks (SAN) were proposed to describe multiple components through the use of automata, whose transitions are determined by local or syncronizing events, having constant or functional rates. Due to the inherent modular representation of SAN, it is possible through tensor (Kronecker) algebra, to store the model infinitesimal generator in memory, in a compact and efficient manner. The numerical methods that calculate the stationary probabilities distribution are adapted to these structured representations.

The basic operation is the vector-descriptor multiplication, which is the product of a probability vector by tensor products composed by sparse matrices. The traditional Shuffle algorithm is characterized by the access and shuffling positions of the vector when multiplied by each matrix of a tensor product term. This approach is considered highly memory-efficient, however, presents a high processing time for the solution of real models. We propose a more flexible and hybrid algorithm for the vector-descriptor product called Split, putting the Shuffle approach in perspective, presenting significant improvements in the execution time for a diverse set of models without impairing the computational resources. Its main idea is to divide each tensorial term in two parts, aggregating its matrices for the calculation of scalars to be tensorly multiplied by the remaining matrices. The algorithm provides gains for the examples, mainly in processing time, even spending more memory.

Nevertheless, increasing the state space of models, this algorithm also becomes unsuitable to obtain a numerical solution. To mitigate the impact of state space explosion, it is proposed the use of simulations to estimate the stationary probability distribution as close as possible to analytical solutions, executing long-run trajectories. We propose the application of perfect sampling techniques (also called exact simulation) to produce reliable samples through trajectory couplings, in reverse simulation time. This technique is distinguished from traditional simulation by avoiding transient periods and the initial state to be chosen. It is discussed the feasibility of these algorithms applied to SAN, specially when monotonicity properties are detected in the models. Partially ordered state spaces allows the execution of an efficient version of the technique by reducing the number of parallel trajectories needed for the generation of a sample.

The iterative numerical analysis and the simulation of stochastic models are approaches that present advantages and limitations when applied to the solution of structured models such as SAN. The main contribution of this thesis focuses on the reduction of the impact of state space explosion of markovian models described in SAN, proposing solutions when the computational time of analytical techniques is too long or when the memory requirements for the probability vector exceeds current technologies storage capacity.

Keywords: Structured Formalisms; Stochastic Automata Networks; Numerical Solutions, Exact Simulation.

Resumo

A solução de modelos markovianos com grande espaço de estados é um dos maiores desafios da área de avaliação de desempenho de sistemas. Os formalismos estruturados, como as Redes de Autômatos Estocásticos (SAN), foram propostos para descrever realidades com múltiplos componentes através de autômatos, cujas transições são regidas por eventos locais ou sincronizantes, com taxas de ocorrência constantes ou funcionais. Devido à capacidade de representação modular de SAN foi possível, através de álgebra tensorial (ou de Kronecker), armazenar o gerador infinitesimal do modelo de forma compacta e eficiente em memória. Os métodos numéricos de solução que calculam a distribuição estacionária das probabilidades são adaptados a estas representações tensoriais.

A operação básica e' a multiplicação vetor-descritor, que é o produto de um vetor de probabilidades por termos tensoriais compostos por matrizes normalmente esparsas. O principal algoritmo de multiplicação chama-se Shuffle e é caracterizado pelo acesso e embaralhamento de posições do vetor quando multiplicado pelas matrizes de cada termo. Este método é considerado extremamente eficaz no armazenamento em memória, entretanto apresenta um tempo de processamento alto para a solução de modelos reais, sendo suas otimizações alvo de pesquisas recentes. Propõe-se um algoritmo híbrido e mais flexível para a multiplicação vetor-descritor, chamado Split, que coloca o algoritmo Shuffle em perspectiva, apresentando ganhos significativos no tempo de execução para diversas classes de modelos, sem onerar os recursos computacionais. Sua ideia principal é dividir cada termo tensorial em duas partes, de forma a agregar algumas de suas matrizes para obtenção de escalares a serem tensorialmente multiplicados pelas matrizes restantes. O uso do algoritmo, dentro de limites gerenciáveis de memória, proporciona ganhos significativos em tempo de processamento, fato demonstrado através de exemplos.

Entretanto, quando os modelos aumentam em escala, este algoritmo torna-se inadequado devido à explosão do espaço de estados. Para mitigar o impacto deste problema propõe-se o uso de soluções alternativas de simulação, as quais buscam estimar a distribuição estacionária de probabilidades tão próximas quanto possível das soluções analíticas, baseando-se na execução de longas trajetórias. Utiliza-se a técnica de simulação baseada em amostragem perfeita (também chamada de simulação exata), onde os algoritmos objetivam fornecer amostras confiáveis da distribuição estacionária através do casamento de trajetórias sobre o espaço atingível, em tempo de simulação reverso. Esta difere-se da simulação tradicional por evitar o período transiente e a escolha aleatória de um estado inicial. Mostra-se a viabilidade destes algoritmos aplicados a SAN, principalmente quando se detectam propriedades de monotonicidade nos modelos. Espaços de estados parcialmente ordenados permitem a execução de uma versão eficiente da técnica ao reduzir o número de trajetórias em paralelo necessárias para obtenção de uma amostra.

A análise numérica iterativa e a simulação de modelos estocásticos são abordagens que apresentam vantagens e limitações quando aplicadas à solução de modelos estruturados como SAN. A principal contribuição desta tese foca na redução do impacto da explosão do espaço de estados de modelos markovianos descritos em SAN, propondo soluções quando o tempo de computação das técnicas analíticas é muito longo ou quando os requisitos de armazenamento do vetor de probabilidade excedem a capacidade de memória das tecnologias correntes.

Palavras-chave: Formalismos estruturados; Redes de Autômatos Estocásticos; Soluções Numéricas; Simulação Exata.

List of Figures

List of Tables

List of Algorithms

Contents

Chapter 1

Introduction

Performance evaluation of modern systems becomes a challenging problem due to the complexity involved in describing and solving models. Several solution techniques are available in the literature but one of the most commonly used techniques is the analytic modeling and evaluation, which produces accurate results. Markov chain is the analytic modeling formalism most widely applied in different domains such as computer, inventory and manufacturing systems, communication networks, bioinformatics, and many other fields not necessarily computational.

The Markov chains formalism power is the simplicity of description because one only needs to characterize a system as discrete states describing the manner in which it moves from one state to another. So the system can be represented as a Markovian process when the time spent in each state appears exponentially distributed. A collection of states is associated to this Markovian process. The system can assume only one state at any time, in other words, the evolution of the process depends exclusively on the current state [61].

Formally, a Markov chain model has a $\mathcal{X}$ finite state space set composed by $n = |\mathcal{X}|$ states, where its transition rates will derive an infinitesimal generator represented by an $n \times n$ square matrix $\mathcal{Q}$, where $\forall_{i \neq j} \ q_{(i,j)} \geq 0$ and $\forall_i \ q_{(i,i)} = -\sum_{j \neq i} \ q_{(i,j)}$. The matrix $\mathcal{Q}$ can be computationally stored in a compact format since it is normally composed by few nonzero elements. In other words it is numerically represented by an unstructured component, *i.e.*, by a unique huge matrix.

The solution of a Markov chain is the stationary probabilities associated to each state in the model and these are often obtained by the long-run probability distribution calculated by an iterative solution for the linear system $\pi\mathcal{Q} = 0$. The probability vector π of dimension n is distributed considering that $\sum_{i=1}^{n} \pi_{(i)} = 1$. Iterative methods [59] such as *Power* method, *Arnoldi* or the generalized minimal residual method (usually abbreviated *GMRES*), compute a successive product of a probability vector π by a Markovian infinitesimal generator $\mathcal{Q}$ (a huge sparse matrix) until it reaches a stationary regime.

Markov chains are solved with a fast computational technique if both matrix and vectors fit in

memory, otherwise the state space explosion will avoid the explicit mapping of the chain to its correspondent transition matrix in memory. The systems of linear equations of this size cannot be solved in practice even with our current technologies. The increasing size and complexity of systems quickly invalidate the use of Markov chains for stochastic modeling of complex systems.

An alternative method to the numerical analysis of models is the simulation based on systems dynamics. An evolution of states is simulated using pseudo-random events to decide the trajectory and from this, the results related to π are estimated. The simulative analysis of systems generates samples from the stationary distribution and it is applicable mainly to large state spaces which can be described by state transition functions based on discrete events [58].

Nevertheless, one must consider the accuracy of results when choosing a given analysis technique, analytic or simulative. The simulation results are statistical in nature, then very long simulation runs are necessary to obtain results with sufficient confidence. There are new simulation techniques concerned in producing exact samples. However, the memory to accomplish this task is considerably increased when applied to huge Markov chains [43].

1.1 Modeling Structured Representations

Not directly applying simulation approaches to solve huge models, structured formalisms based on Markov chain principles introduce the possibility of describing more than one irreducible component, with interactions among components and individual behavior. This interactions are called events, which are primitives associated by rates or probabilities. Due to the fact that systems are normally composed of many components, Markovian structured formalisms like Stochastic Petri nets (SPN) [1], Stochastic Automata Networks (SAN) [55] and Performance Evaluation Process Algebras (PEPA) [44] were proposed to cope with the problem of the state space explosion and the consequent matrix storage problems.

Given the difficulty in constructing these models, the use of SAN, proposed by Plateau [54], is becoming increasingly important in the field of stochastic modelling of parallel and distributed computer systems, such as communicating processes, concurrent processors, shared memory, behavior of communication network protocols, and many other realities inside the scope of Markov chain applications [4, 8, 12, 20, 29, 34, 35, 53, 57]. The advantage of SAN modeling over other formalisms is its simplicity and similarity with Markov chains in terms of the finite number of states and transitions labeled by events.

The specific advantage the SAN approach has over generalized stochastic Petri nets, and indeed over any Markovian analysis that requires the generation of a transition matrix containing rates, is that its representation remains compact even as the number of states in the underlying Markov chain

begins to explode. The state transition matrix is not stored, instead, it is represented by a number of much smaller matrices. These matrices are defined with all relevant information may be determined without explicitly generating the global matrix. This research field presents many alternative modeling paradigms, but structured models have the common need of a compact representation, an efficient storage and alternative solutions for very large state spaces.

1.2 Solutions for Kronecker-based Descriptors

An available alternative to a compact storage for SAN models is the implicit generation of the infinitesimal generator $\mathcal{Q}$ of the underlying Markov chain, keeping the structured characteristics into their internal representation. This advance becomes possible employing tensor (Kronecker) algebra to store the infinitesimal generator, so large systems can be represented by a sum of K tensor products of N matrices given algebraically by $\sum_{j=1}^{K}(\bigotimes_{i=1}^{N}\mathcal{Q}_j^{(i)})$. The sparse matrices composing each tensor product are in fact a way of taking advantage of all the structural information inside original automata. Such principle appears since the first definitions of SAN [55], but recently it has also been used in other stochastic formalisms [32, 45]. In all those references, the term *descriptor* is used to refer to a tensor represented infinitesimal generator.

Another intrinsic aspect related to structured formalisms is the insertion of unreachable states when computing the probability vector π so many approaches use Kronecker algebra* for representation [21, 23, 52]. Their solution have implicit tensor operations called vector-descriptor product and the method is based on *shuffle* algebra principles [26]. The complexity to treat memory-efficient structured models lead us to a less time-efficient solution approach when compared to the pure sparse multiplication, which can be easily done respecting the memory bounds. For further information about tensor algebra properties and their application in specialized numerical algorithms refer to [2, 26, 37].

The current SAN solver use a Kronecker description as basis to perform vector-descriptor multiplications inside the iterative methods implemented. The *Shuffle* algorithm is known as the most memory-efficient numerical solution for descriptors [6, 16, 37]. The tensor product operation is summarized as $\sum_{j=1}^{K}\left[\pi(\bigotimes_{i=1}^{N}\mathcal{Q}_j^{(i)})\right] = 0$, where π is a probability vector and $\bigotimes_{i=1}^{N}\mathcal{Q}_j^{(i)}$ is the j^{th} tensor product in a *descriptor*. Note that the descriptor size K can be potentially increased when synchronizations become more frequent among model components. This algebraically represents more tensor products inside descriptors, consequently the total number of iterations become more time consuming until convergence.

Moreover, there is still a memory limit imposed by the current technology even when dealing

*The basic concepts of the classical tensor algebra are detailed on the Appendix B of this thesis.

with structured descriptors. The state space explosion in this case can avoid the application of the traditional numerical solution. Several approaches are proposed to deal with massive product state spaces, the techniques vary from hybrid solutions for simulation and numerical analysis [15] to pure forward simulations [10, 46, 47].

The Markovian simulation research has evolved to be applied to situations not necessarily involving any time dynamics in the chains. The problem is restricted to generate samples directly from the stationary distribution. The transition kernel of a Markov chain of $n = |\mathcal{X}|$ states allows to start a trajectory from an initial state $s_0 \in \mathcal{X}$ chosen arbitrarily, running for a time t and outputting the state $s_t \in \mathcal{X}$. The forward simulation approaches execute a fixed number of steps walking in the chain to collect samples. The major problem of forward simulations is the definition of an initial state and the quantity of steps to execute, *i.e.*, to decide how many steps are considered satisfactory to avoid the transient period of the these simulations. The uncertainty of these parameters generates bias samples.

Approaches based on *Coupling from the Past* techniques [56] (also called *perfect sampling* or *exact simulation*) emerge to guarantee samples confidence in Markovian simulations [65]. These techniques consider all states as initial states, running trajectories in parallel until their coalescence[†]. It is proved in the literature [43, 51, 56] that the coupling in backward steps guarantee unbiased samples. The perfect sampling technique can be applied to solve many systems, mainly those with huge state spaces and an identified monotone set of events. There are recent researches conducted towards algorithmical adaptation for monotone systems [66], for example, reducing a huge number of initial states to control in parallel into selected trajectories based on monotonicity properties.

Simulative techniques are considered alternative methods for the solution of structured descriptors where the computational resources are insufficient to perform an analytical solution. The challenge in this research is related to the adaptation of these current simulation advances to complex structures with huge underlying state spaces.

1.3 Thesis Objective

This thesis enforces that numerical analysis and simulation both have advantages and limitations when applied to the solution of structured models such as stochastic automata networks. Note that an analytical solution of models is always the best alternative when accuracy is needed. However, for all other models we could not even generate the state space (for example those underlying a huge queueing network or a SAN) we demonstrate that it is possible to design adapted exact simulation algorithms. The available numerical algorithms for the SAN formalism are basically iterative solvers for Kronecker representations [37] (set of transition matrices operating with tensor algebra),

[†]The term *coalescence* in this thesis is also referred as *coupling* of trajectories.

implemented in the *PEPS* software tool environment [5, 49], and some first approaches to simulate SAN focusing on the concept of uniformized events and forward simulation techniques [57].

The objective of this thesis is the reduction of the impact of state space explosion in the solution of huge models described as Stochastic Automata Networks.

The state space explosion is the major problem of the analytical modeling and its numerical solution. This thesis focus is the numerical solution of models described as sets of stochastic automata and the contribution is two folded: (i) provides a Kronecker-based descriptors solution which aims to speed up the vector-descriptor product and (ii) provides an alternative solution for event-based descriptors based on advanced simulation techniques such as the *exact simulation*. Both approaches take advantage of the structural configuration of models to optimize their operations.

The Kronecker based approach helped changing the compositional viewpoint towards efficient solutions. However, despite the positive theoretical points, the impact of the tensor algebra for the study of models coming from the real world is still very limited [16, 33]. Many researches focused on a memory-efficient approach without perceiving that the formal restrictions in this case lead us many times to time-inefficient solutions. We conclude that is possible to design algorithms to consider the aggregation or splitting selected sparse parts of descriptors, consequently accelerating the vector-descriptor product. The objective is not to devise a method to store also the vector in an efficient manner, but to multiply this (huge) vector in a time-efficient manner when compared to the *Shuffle* approach.

Additionally, for those vectors and descriptors that can not fit in memory, we propose an approximated solution based on *perfect sampling*, with a memory-efficient approach when monotonicity properties can be applied. Many researches were already conducted proposing new storage techniques associated with specialized algorithms for structured models in general, not specifically for SAN. Nevertheless, alternative solutions such as simulations approaches were not deeply studied in the context of SAN. We conclude that is possible to adapt backward coupling algorithms to obtain unbiased samples for statistical analysis. Consequently, there is a need in SAN solution research to devise methods that exploit monotonicity properties of models and potentially other structural information.

The research presented in this thesis points out new solutions for the SAN formalism, which in future works can possibly be generalized to any structured model representation, in which systems are described by independent components, and each one can have interdependencies given by synchronizing or functional transitions [54, 55].

The next section details the specific research guidelines to accomplish the objective of this thesis.

1.3.1 Hybrid numerical algorithms to reduce computational time

This section describes the first axis of this thesis which is the improvement of the computational time of the vector-descriptor product, maintaining the solution efficacy. Tensor products related to SAN models of practical applications are quite sparse mainly because they represent dependent behaviors among automata considering each event separately. These characteristics lead us to analyze the possibility of taking advantage of matrices sparsity (already indicated as an advantage for numerical methods [16]), combined with the natural sparsity of classical tensor product decomposed in normal factors [37]. Considering the classical Kronecker properties used to decompose tensor products, we design a flexible and hybrid approach to the vector-descriptor multiplication, exploiting the tradeoff between time and memory to propose a new algorithm called *Split*.

In fact, the *Split* algorithm proposes a way to balance the actual memory cost to reduce the multiplications needed to complete an iteration, resulting in a more time-efficient approach for many cases, where the matrices composing descriptors are sparse or ultra-sparse[‡]. Even so we are obliged to augment the memory cost storing new structures, it is reduced the vector-descriptor product complexity. In the worst case, *Split* has the same order of complexity presented by the traditional *Shuffle* method.

The new algorithm is analyzed in terms of its efficacy to solve models and also efficiency when compared to the traditional *Shuffle* method for Kronecker-based descriptors. Moreover, a set of numerical optimizations are proposed to accelerate even more this complex, but necessary, operation. Although its flexibility to deal efficiently with classical Kronecker descriptors, the product state space is still a limit to solve huge models even with advanced solutions for linear equations. Note that this hybrid solution is applied to solve descriptors using classical tensor algebra. Considering that models with a generalized descriptor can be rewritten with only constant values in the matrices, then functional dependencies are not discussed in the context of this thesis. Nevertheless, results obtained with constant rates, show that once we have a time-efficient way to evaluate measures of interest, the *Split* algorithm will continue to be as efficient as it already is. In the conclusions, we point out some future works in the direction of a functional hybrid solution between sparse techniques and generalized Kronecker descriptors.

1.3.2 Advanced simulation techniques to reduce memory costs

This section discusses the second axis of this thesis which is the application of advanced simulations in the context of SAN. A new algorithmic solution called *perfect sampling* or *exact simulation* which is based on backward coupling as the *Coupling from the Past* (CFTP) algorithm [56] overcome

[‡] A matrix classification in sparse or ultra-sparse is given by the relation of the total number of nonzero elements by its dimension [16].

the burning time problem generating unbiased samples. CFTP proposes the execution of trajectories in parallel, starting from all states of the Markov chain. Running time in backward steps, the coupling of all trajectories in a given state guarantees that this sample is originated from the stationary regime. Researches on perfect simulation of Markovian queueing networks [65, 66, 63, 64] shows possible algorithmic improvements when the state space has a partial ordering.

The proposition of an exact simulation application to SAN opens new discussions about structured models,allowing the description of an even more complex variety of systems. Another contribution of this thesis is the study of the partial ordering of product state spaces, concluding that for some models with a component-wise ordering it is possible to run alternative solutions [38]. We propose one adaptation of perfect sampling techniques to solve SAN models, regarding their reachable state space, also some structural analysis to devise monotone versions of the backward coupling method.

The simulation of SAN can take advantage of the underlying chain structural properties and indexes of interest, to find a way to solve with less memory costs. We work on the memory drawback imposed by backward coupling techniques contracting the reachable state space in a smaller subset. In this work, we focus on the SAN adaptability to exact simulation to produce model evaluations, *e.g.*, performance indexes, steady-state probabilities. The research of time-efficient simulations is out of the scope of this thesis, but some considerations about improvements of this issue are drawn on the conclusions.

1.4 Thesis Organization

This thesis is composed of four chapters, a conclusion and three appendices. Firstly we explained the background related to the numerical solution of Markovian models focusing on SAN as modeling formalism. The solutions proposed are presented in different chapters and appendices. This introduction explained the major objective of this thesis as well as the research directions to accomplish it.

Chapter 2 focuses on SAN descriptor to allow the comprehension of developed models and numerical solutions proposed in this thesis. Forward in the context of a discrete-event description, this chapter also extends the definition of well-defined SAN, *i.e.*, the reachable state space is compatible with the system evolution rules.

Chapter 3 explains how a SAN model can be structurally explored to obtain the solution combined with Kronecker algebra to efficiently store the descriptor. Different SAN models are used to validate the hybrid algorithm proposed (resource sharing and allocation models, a model for wireless networks protocol evaluation and a model for a parallel algorithm implementation) collecting both execution times and memory spent also for the traditional *Shuffle* method.

Chapter 4 shows that is possible to design a perfect simulation algorithm for SAN, *i.e.*, using

backward coupling simulation to sample states directly from the stationary distribution. We briefly review simulation concepts, and using specific SAN examples, we show what is structurally fitted to run memory-efficient backward simulations, *e.g.*, a canonical or component-wise structure[§] (queueing systems and resource allocation models such as those based on the classical *dining philosophers* problem).

Chapter 5 is the conclusion and presents a brief summary of the results obtained in the context of this thesis, in comparison with the solution bounds pointed out in the literature. This comparison shows the effectiveness demonstrated in the practical results of the advanced numerical method proposed, and the theoretical contribution of a simulation technique presented as an alternative SAN solution based on the coupling method. Moreover, this chapter presents final considerations about the future works, prioritizing the solution of huge state spaces.

Appendix A presents a description of the SAN examples studied in the Chapters 3 and 4. The Appendix B is a review of classical tensor algebra properties and the tensor product decomposition in normal factors needed in the Chapters 2 and 3. Appendix C presents the notation used in each chapter of this thesis, indexed by section.

[§]Structure classification, as *canonical* or *component-wise*, in the context of this thesis is related to the chain structure underlying a SAN model.

Chapter 2

Stochastic Automata Networks

The first step involved in calculating the steady-state distribution of discrete systems is to characterize the states of the system and describe the manner in which it moves from one state to another [61]. Given the difficulty in constructing these models as a unique component, the use of stochastic automata networks (SAN) as proposed by Plateau [54] is becoming increasingly important for modeling parallel and distributed systems such as communicating processes, concurrent processors, shared memory, behavior of communication network protocols and many other applications non specific to computer science. However the complexity of the modeling process brings the need of advanced numerical solutions.

Structured formalisms such as SAN allow modeling with more than one component that operate more or less independently, requiring sometimes interactions such as synchronizing their actions, or operating at specific rates (or probabilities) depending on the state of other component. The concept of synchronizing events and functional dependencies are introduced with SAN as powerful primitives to represent different realities.

2.1 Basic Concepts and Definitions

The SAN basic idea is to represent a whole system by a collection of K subsystems described as K stochastic automata $\mathcal{A}^{(k)}$, with $k \in [1..K]$. In each of these automata the transitions are labeled by events. Each event includes probabilistic and timing information, and the network of automata has a set ξ with all events in the model. This framework defines a modular way to describe continuous and discrete-time Markovian models [54, 55].

Definition. Each automaton $\mathcal{A}^{(k)}$ has a set $\delta^{(k)}$ of local states $s^{(i)}$ where $i \in \{1 \ldots n_k\}$, interconnected by transitions and their respective events $e_p \in \xi$, $\xi = \{e_1, \ldots, e_P\}$ considering P events in the

model. The constant n_k is the cardinality of $\delta^{(k)}$, *i.e.*, the total number of states in automaton $\mathcal{A}^{(k)}$. Each event $e_p \in \xi$ has its own rate λ_p and a probability associated.

Definition. The local state $s^{(i)}$ ($i \in \{1 \dots n_k\}$) of an automaton $\mathcal{A}^{(k)}$ is just the state it occupies at a time t.

Definition. A global state $\tilde{s}$ of a SAN model with K automata is a vector $\tilde{s} = \{s^{(1)}; \dots; s^{(K)}\}$ where each automaton $\mathcal{A}^{(k)}$ is in the local state $s^{(k)} \in \delta^{(k)}$ at a time t.

Definition. The set of all global states is called *product state space*. The product state space $\mathcal{X}$ of a SAN model is the Cartesian product of all sets $\delta^{(k)}$.

It is natural the insertion of unreachable states inside the product state space since not all combinations of local states are valid in the global perspective. The reachable state space indicates the consistent state space related to a model, *i.e.*, all the states inside this set are reachable by any other state.

Definition. The reachable state space $\mathcal{X}^R_{\tilde{s}_0}$ (or $\mathcal{X}^R$) is an irreducible component obtained from a given initial global state $\tilde{s}_0 \in \mathcal{X}$ and successive firing of events in ξ. Each global state $\tilde{s}$ reached by any possible combination of events is included in this set.

A SAN model can describe a reachability function $\mathcal{F}^R$ which indicates the global states effectively accessible by one or more events fired composing $\mathcal{X}^R$.

Definition. The reachability function $\mathcal{F}^R$ is considered well-defined if and only if it indicates exactly the states in the set $\mathcal{X}^R$, *i.e.*, the component $\mathcal{X}^R$ presents a strong connected transition graph.

The event e_p can occur in more than one automata meaning there is a synchronization occurring among components, *i.e.*, the event e_p is associated to more than one transition in different automata. An event e_p can also be local in one automaton, *i.e.*, it occurs only inside a component in a given local state $s^{(k)}$. Sometimes from this local state $s^{(k)}$, the occurrence of an event e_p can lead to more than one state, *i.e.*, there is more than one transition labelled by the same event. To deal with this, one additional *probability* is associated to e_p in each transition. The absence of probabilities in transitions is tolerated if only one transition can be fired by an event from a given local state.

Events can contain constant or functional rates, in the functional case, the event can be dependent of the local state of other automata to be fired as synchronizing events do, without changing every local automaton in the function. Note that the importance of modeling primitives such as synchronizing events and functional rates is the facility to model characteristics that conveys real world problems, where components always interact in some manner beyond their individual behavior. Functional interactions can be also represented by synchronizing events [6, 13] since it is possible to set a state

with a *self*-transition, *i.e.*, the local state can fire a transition without really changing its local state.

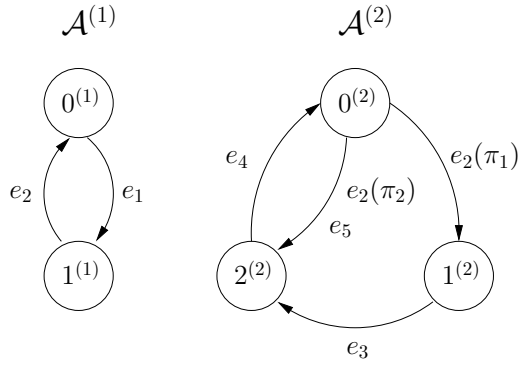
Definition. A SAN model can be called *well-formed* if and only if the $\mathcal{X}^R$ component is irreducible.

A *well-formed* model contains states that are reachable by any other state and states that fire some transition to at least one another state, which means there is no absorbent neither unreachable state. In this thesis, we assume that the reachable set is given by the model designer. Despite of that, it is relevant to numerical solutions that the reachable state space $\mathcal{X}^R$ is, in the worst case, equal to the product state space $\mathcal{X}$ of the SAN model, *i.e.*, $\mathcal{X}^R \subseteq \mathcal{X}$.

The large models we are interested, in the context of this work, are those with a huge $\mathcal{X}$ and fewer reachable global states to deal with. A great advantage from this fact is that, knowing $\mathcal{X}^R$, one can reduce the overhead related to the state space explosion problem adapting this characteristic inside the numerical solution as an optimizing factor. The efficient generation and storage of reachable states are not in the context of this work, mainly because to cope with that there are very efficient approaches already studied [17, 52].

2.2 Graphical Representation and Primitives

A SAN model presents a simple graphical representation and this is a great feature considering the modeling of complex behaviors. In this section is presented an example with the available modeling primitives. Figure 2.1 is a SAN model with two automata $A^{(i)}$, and five events ($|\xi| = 5$).



Type	Event	Rate
loc	e_1	f_1
syn	e_2	α_2
loc	e_3	α_3
loc	e_4	α_4
loc	e_5	α_5

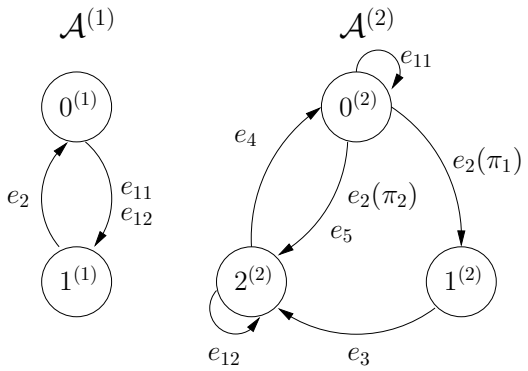
$$f_1 = \left[\left(st\ \mathcal{A}^{(2)} == 0^{(2)}\right) * \alpha_{11}\right] + \left[\left(st\ \mathcal{A}^{(2)} == 2^{(2)}\right) * \alpha_{12}\right]$$

Figure 2.1: Example of a SAN model with a functional rate

This example shows two types of events (local and synchronizing) with constant rates (represented by α_i) and functional rates (represented by f_1). The functional rate f_1 returns a constant value

considering evaluations of other automaton local states using the primitive *st, i.e.*, the associated event is fired with a function that depends on the local states of other automata. Events that occur in different transitions issued from the same state must present a probability of occurrence associated to the rate. In the example, the event e_2 with constant rate α_2 is associated to the probabilities π_1 and π_2 respectively.

The model presented on the Figure 2.1 has a functional dependency that could also be expressed as synchronizing events with constant rates associated. The conversion of a descriptor based on generalized tensor algebra in a descriptor using classical tensor algebra is a process already studied [13] and is possible to achieve an equivalence of results* using different modeling primitives. The descriptors containing functional rates are converted to a classical algebra representation. Figure 2.2 represents a classical description of the example in Figure 2.1.



Type	Event	Rate
syn	e_{11}	α_{11}
syn	e_{12}	α_{12}
syn	e_2	α_2
loc	e_3	α_3
loc	e_4	α_4
loc	e_5	α_5

Figure 2.2: SAN model of Figure 2.1 without functional rates

For both representations we have an equivalent Markov chain representing the reachable state space $\mathcal{X}^R$ of the model, where all states, but one in $\mathcal{X}$, are reachable. Figure 2.3 shows all states and transitions for the example, indicating the correspondent Markov chain (the $\mathcal{X}^R$ set) in the inner box.

SAN descriptions can define the $\mathcal{X}^R$ set through the insertion of a reachability function $\mathcal{F}^R$ by the designer. The boolean evaluation of this function, when applied to every global state inside $\mathcal{X}$, returns the reachable states composing $\mathcal{X}^R$. One can also indicate a subset of $\mathcal{X}^R$ using a *partial reachability* function denoted by $\mathcal{F}^{R*}$, where it is possible to indicate at least one reachable state for the model (so the set can be derived from this state or subset of states). More details can be found in [5, 49].

The example presents only one unreachable state, so the reachability function $\mathcal{F}^R$ can be defined

*The impact of functional rates in comparison to the use of synchronizing events [13] are not explored in this thesis.

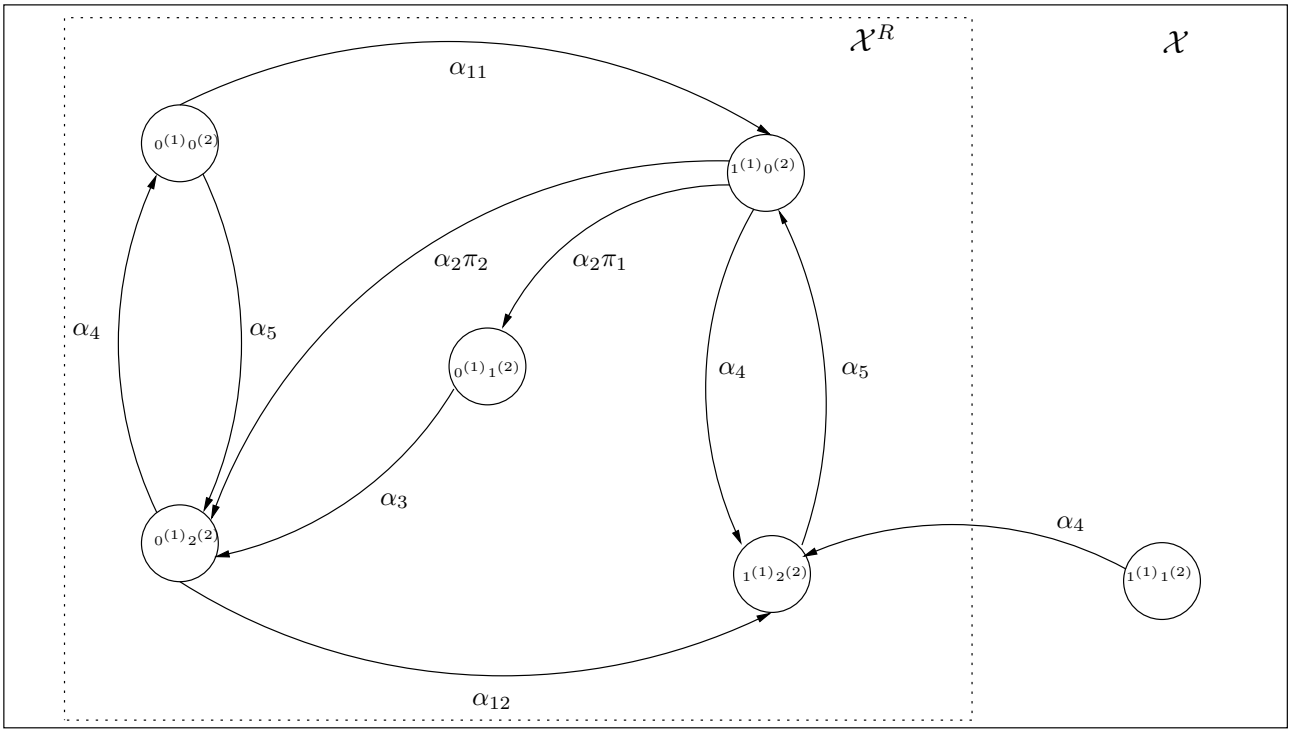


Figure 2.3: Equivalent Markov chain for both SAN examples (Figure 2.1 and 2.2)

to exclude the global state $\{1^{(1)};1^{(2)}\}$ as following:

$$\mathcal{F}^R = (\text{st } A^{(1)} \; != \; 1^{(1)}) \;\&\&\; (\text{st } A^{(2)} \; != \; 1^{(2)});$$

The function $\mathcal{F}^R$ can also be partially defined with only a subset of reachable states, then it can be later calculated iteratively from these specific states. As an example, the global state $\{0^{(1)};0^{(2)}\}$ is indicated in the partial reachability function $\mathcal{F}^{R*}$:

$$\mathcal{F}^{R*} = (\text{st } A^{(1)} \; == \; 0^{(1)}) \;\&\&\; (\text{st } A^{(2)} \; == \; 0^{(2)});$$

For huge models is really difficult to determine the fully function $\mathcal{F}^R$ in a model description. There are researches concerned in propose very efficient approaches for the determination and for the compact storage of the reachable set [17, 52]. In the context of SAN formalism, the definition of a partial reachability function $\mathcal{F}^{R*}$ is sufficient for the current PEPS tool [5, 49] to find the states in $\mathcal{X}^R$. The following section discusses the internal structural representation of SAN models presenting two different approaches despite both use the concept of events as a key for enabling new numerical solutions.

2.3 Structural Representations

Given a well-formed SAN description which is a structured representation of a Markov chain, the next step is to store and solve the model analytically (or numerically) to obtain the stationary (or transient) probability distribution. Following Kronecker properties (refer to Appendix B) the modular behavior of the system can be decomposed in a *descriptor*, *i.e.*, a set of small transition matrices containing the model rates as elements. This decomposition allows iterative solvers to perform the vector-matrix product, considering a multidimensional state space.

However, for SAN models, or even for other structured formalisms, there is another possible structured representation which can be simply event-based. We can extract the set of events related to a model (uniformizing their respective rates) and construct a transition function to represent the firing of events. Both approaches must deal with a SAN model (or any structured model) as input, and become a basis to generate the probability distribution as output. In this section, we briefly discuss separately both approaches of SAN descriptions: *Kronecker*-based and *Event*-based.

2.3.1 Kronecker-based *descriptor*

The use of tensor (Kronecker) algebra [2, 26, 37] to represent large complex models in a structured description undeniably reduces the needs of memory avoiding the storage of the infinitesimal generator which is a full flat matrix. The SAN formalism, for example, takes advantage of this approach to represent the infinitesimal generator of the model as an algebraic formula. In fact, instead of defining one single, and usually huge, matrix of order equal to the product state space of the model, we define a set of tensor product terms with smaller matrices in which the combination through Kronecker operations is equal to the underlying Markov chain transition matrix.

The local behavior of each K automaton is represented by a unique tensor sum of K matrices. We indicate by $\mathcal{Q}_l^{(k)}$ the matrices composed of local events occurrence rates as elements, with a correspondent negative diagonal adjustment, which is a negative value correspondent to the line ordinary sum. The synchronizing behavior is represented by tensor product terms of matrices containing the rates of these synchronizing events. For each event is generated a tensor product term where one affected matrix called $\mathcal{Q}_{e_p{}^+}^{(k)}$, contains the occurrence rate, and the other matrices involved contain the value 1 in the respective synchronizing transition. Automata that not interfere the synchronization, present identity matrices $I_{\mathcal{Q}^{(k)}}$ of order n_k in the tensor product term.

For each tensor product term generated by a synchronizing event e_p (positive tensor term $e^+ = \bigotimes_{k=1}^{K} \mathcal{Q}_{e_p^+}^{(k)}$), another related tensor term is needed containing the diagonal adjustment for the synchronizing behavior with matrices $\mathcal{Q}_{e_p{}^-}^{(k)}$ (negative tensor term $e^- = \bigotimes_{k=1}^{K} \mathcal{Q}_{e_p^-}^{(k)}$). Here the number of synchronizing events is given by E.

The *descriptor* is then composed of two parts and is given by the Equation 2.1:

$$\mathcal{Q} = \bigoplus_{k=1}^{K} \mathcal{Q}_l^{(k)} + \sum_{e_p \in E} \left(\bigotimes_{k=1}^{K} \mathcal{Q}_{e_p^+}^{(k)} + \bigotimes_{k=1}^{K} \mathcal{Q}_{e_p^-}^{(k)} \right) \tag{2.1}$$

Kronecker algebra properties (see Appendix B) decompose a tensor sum in an ordinary multiplication of tensor products called normal factors [55], for example, $\mathcal{Q}^{(1)} \oplus \mathcal{Q}^{(2)} = (\mathcal{Q}^{(1)} \otimes I_{\mathcal{Q}^{(2)}}) \times (I_{\mathcal{Q}^{(1)}} \otimes \mathcal{Q}^{(2)})$. Consequently, a *descriptor* $\mathcal{Q}$ can be defined as a sum of only tensor products of matrices generically expressed by $\mathcal{Q}_j^{(k)}$.

Definition. Each matrix $\mathcal{Q}_j^{(k)}$ has its dimension given by the the cardinality of $\delta^{(k)}$ (n_k) of each automaton $\mathcal{A}^{(k)}$ in the model. Their elements are the rates (or probabilities) associated to each transition in $\mathcal{A}^{(k)}$ depending on the tensor product been analyzed.

Definition. A SAN model with K automata and E synchronizing events has as *descriptor* $\mathcal{Q}$ as an algebraic formula composed by a sum of $K + 2E$ tensor products with K matrices $\mathcal{Q}_j^{(k)}$ each. Algebraically, $\mathcal{Q} = \sum_{j=1}^{K+2E} \bigotimes_{k=1}^{K} \mathcal{Q}_j^{(k)}$

Table 2.1 details the descriptor $\mathcal{Q}$ showing the tensor sum operation in the local part decomposed into a ordinary sum of K normal factors, *i.e.*, a sum of tensor products where all matrices but one are identity matrices. Therefore, only the non-identity matrices (in the local part the matrices $Q_l^{(k)}$) need to be stored. The synchronizing part of the *descriptor* is represented with positive tensor terms e^+ and negative tensor terms e^-.

For the SAN example in Figure 2.2, the *descriptor* is a set of $K+2E$ tensor products and follows the general descriptor formula (considering $K = 2$ automata and $E = 3$ synchronizing events):

$$\begin{aligned}
\mathcal{Q} = \sum_{j=1}^{K+2E} \bigotimes_{k=1}^{K} \mathcal{Q}_j^{(k)} = \quad & (\mathcal{Q}_l^{(1)} \otimes I_{\mathcal{Q}_l^{(2)}}) + (I_{\mathcal{Q}_l^{(1)}} \otimes \mathcal{Q}_l^{(2)}) + \\
& (\mathcal{Q}_{e_{11}^+}^{(1)} \otimes \mathcal{Q}_{e_{11}^+}^{(2)}) + (\mathcal{Q}_{e_{11}^-}^{(1)} \otimes \mathcal{Q}_{e_{11}^-}^{(2)}) + \\
& (\mathcal{Q}_{e_{12}^+}^{(1)} \otimes \mathcal{Q}_{e_{12}^+}^{(2)}) + (\mathcal{Q}_{e_{12}^-}^{(1)} \otimes \mathcal{Q}_{e_{12}^-}^{(2)}) + \\
& (\mathcal{Q}_{e_2^+}^{(1)} \otimes \mathcal{Q}_{e_2^+}^{(2)}) + (\mathcal{Q}_{e_2^-}^{(1)} \otimes \mathcal{Q}_{e_2^-}^{(2)})
\end{aligned}$$

Note that the automaton $\mathcal{A}^{(1)}$ has no local events so the local part of the formula has in fact $K-1$ normal factors to decompose, *i.e.*, there is no factor $\mathcal{Q}_l^{(1)} \otimes I_{\mathcal{Q}_l^{(2)}}$ in the decomposition of $\mathcal{Q}_l^{(1)} \oplus \mathcal{Q}_l^{(2)}$, we have just $I_{\mathcal{Q}_l^{(1)}} \otimes \mathcal{Q}_l^{(2)}$ to analyze. For each synchronizing event we proceed with the decomposition in positive e^+ (and negative e^-) tensor products, summarizing $2E$ terms. Following are presented the matrices composing each tensor term for the SAN example in Figure 2.2.

$\sum$	K		$Q_l^{(1)} \otimes I_{\mathcal{Q}^{(2)}} \otimes \cdots \otimes I_{\mathcal{Q}^{(K-1)}} \otimes I_{\mathcal{Q}^{(K)}}$ $I_{\mathcal{Q}^{(1)}} \otimes Q_l^{(2)} \otimes \cdots \otimes I_{\mathcal{Q}^{(K-1)}} \otimes I_{\mathcal{Q}^{(K)}}$ $\vdots$ $I_{\mathcal{Q}^{(1)}} \otimes I_{\mathcal{Q}^{(2)}} \otimes \cdots \otimes Q_l^{(K-1)} \otimes I_{\mathcal{Q}^{(K)}}$ $I_{\mathcal{Q}^{(1)}} \otimes I_{\mathcal{Q}^{(2)}} \otimes \cdots \otimes I_{\mathcal{Q}^{(K-1)}} \otimes Q_l^{(K)}$
	$2E$	e^+	$Q_{e_1^+}^{(1)} \otimes Q_{e_1^+}^{(2)} \otimes \cdots \otimes Q_{e_1^+}^{(K-1)} \otimes Q_{e_1^+}^{(K)}$ $\vdots$ $Q_{e_E^+}^{(1)} \otimes Q_{e_E^+}^{(2)} \otimes \cdots \otimes Q_{e_E^+}^{(K-1)} \otimes Q_{e_E^+}^{(K)}$
		e^-	$Q_{e_1^-}^{(1)} \otimes Q_{e_1^-}^{(2)} \otimes \cdots \otimes Q_{e_1^-}^{(K-1)} \otimes Q_{e_1^-}^{(K)}$ $\vdots$ $Q_{e_E^-}^{(1)} \otimes Q_{e_E^-}^{(2)} \otimes \cdots \otimes Q_{e_E^-}^{(K-1)} \otimes Q_{e_E^-}^{(K)}$

Table 2.1: SAN *descriptor*

Local behavior of automaton $\mathcal{A}^{(2)}$ represented as a tensor product:

$$I_{\mathcal{Q}_l^{(1)}} \otimes \mathcal{Q}_l^{(2)} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} -\alpha_5 & 0 & \alpha_5 \\ 0 & -\alpha_3 & \alpha_3 \\ \alpha_4 & 0 & -\alpha_4 \end{pmatrix}$$

Behavior of synchronizing event e_{11} with constant rate α_{11}:

- Positive tensor term (${e_{11}}^+$)

$$\mathcal{Q}_{{e_{11}}^+}^{(1)} \otimes \mathcal{Q}_{{e_{11}}^+}^{(2)} = \begin{pmatrix} 0 & \alpha_{11} \\ 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

- Negative tensor term ($e_{11}{}^{-}$)

$$\mathcal{Q}^{(1)}_{e_{11}{}^{-}} \otimes \mathcal{Q}^{(2)}_{e_{11}{}^{-}} = \begin{pmatrix} -\alpha_{11} & 0 \\ 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Behavior of synchronizing event e_{12} with constant rate α_{12}:

- Positive tensor term ($e_{12}{}^{+}$)

$$\mathcal{Q}^{(1)}_{e_{12}{}^{+}} \otimes \mathcal{Q}^{(2)}_{e_{12}{}^{+}} = \begin{pmatrix} 0 & \alpha_{12} \\ 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- Negative tensor term ($e_{12}{}^{-}$)

$$\mathcal{Q}^{(1)}_{e_{12}{}^{-}} \otimes \mathcal{Q}^{(2)}_{e_{12}{}^{-}} = \begin{pmatrix} -\alpha_{12} & 0 \\ 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Behavior of synchronizing event e_2 with constant rate α_2 (associated probabilities π_1 and π_2):

- Positive tensor term ($e_2{}^{+}$)

$$\mathcal{Q}^{(1)}_{e_2{}^{+}} \otimes \mathcal{Q}^{(2)}_{e_2{}^{+}} = \begin{pmatrix} 0 & 0 \\ \alpha_2 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & \pi_1 & \pi_2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

- Negative tensor term ($e_2{}^{-}$)

$$\mathcal{Q}^{(1)}_{e_2{}^{-}} \otimes \mathcal{Q}^{(2)}_{e_2{}^{-}} = \begin{pmatrix} 0 & 0 \\ 0 & -\alpha_2 \end{pmatrix} \otimes \begin{pmatrix} (\pi_1 + \pi_2) & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Many approaches are studied to store and manipulate a Kronecker representation [21, 23, 52]. The descriptor can be stored as a set of sparse matrices using a compact format, for example, Harwell-Boeing format (HBF) [61] where only the nonzero elements and their indexes are explicitly indicated.

The tensor operations are implicit in the vector-descriptor product and the current SAN solver uses this description inside iterative methods. More information on this subject is presented and discussed on Chapter 3.

2.3.2 Event-based *descriptor*

This section proposes another discrete-event representation [42] for SAN models, where the set ξ of events can be defined with an associated transition function Φ between global states. Considering the product state space $\mathcal{X}$ of a model, and the fact that SAN have an underlying Markovian system, we consider the model composed by a set of global states $\tilde{s}$ and a finite collection $\xi = \{e_1, \ldots, e_P\}$ of P events (local and synchronizing events are all in the set ξ).

Definition. The global transition function defined by $\Phi(\tilde{s}, e_p) = \tilde{r}$ ($p \in [1..P]$) is the set of rules that associate to each global state $\tilde{s} \in \mathcal{X}$ a new global state denoted by $\tilde{r} \in \mathcal{X}^R$, through the firing of the transition labeled by event $e_p \in \xi$.

Definition. An event e_p is said to be enabled in the global state $\tilde{s} \in \mathcal{X}$, if and only if $\Phi(\tilde{s}, e_p) = \tilde{r}$, where $\tilde{s} \neq \tilde{r}$, and $\tilde{r} \in \mathcal{X}$. Analogously, an event is said to be disabled in state $\tilde{s}$, if and only if $\Phi(\tilde{s}, e_p) = \tilde{r}$, and $\tilde{s} = \tilde{r}$.

In each global state $\tilde{s}$ some events are enabled, *i.e.*, they change the global state $\tilde{s}$ into another state $\tilde{r}$. However, not all events may occur from a given global state. In those cases, the transition function assigns the permanence in the same global state. Table 2.2 shows the transition function application for the SAN in Figure 2.2, considering all global states $\tilde{s} \in \mathcal{X}^R$ and all events $e_p \in \xi$. In this table, the resulting global states $\tilde{r} = \Phi(\tilde{s}, e_p)$ are represented, where those corresponding to possible transitions are marked in bold face, *i.e.*, those corresponding to enabled events[†].

$\tilde{s} \in \mathcal{X}^{\mathcal{R}}$	$\tilde{r} = \Phi(\tilde{s}, e_p), e_p \in \xi$					
	$\Phi(\tilde{s}, e_{11})$	$\Phi(\tilde{s}, e_{12})$	$\Phi(\tilde{s}, e_2)$	$\Phi(\tilde{s}, e_3)$	$\Phi(\tilde{s}, e_4)$	$\Phi(\tilde{s}, e_5)$
{0;0}	**{1;0}**	{0;0}	{0;0}	{0;0}	{0;0}	**{0;2}**
{0;1}	{0;1}	{0;1}	{0;1}	**{0;2}**	{0;1}	{0;1}
{0;2}	{0;2}	**{1;2}**	{0;2}	{0;2}	**{0;0}**	{0;2}
{1;0}	**{0;1}**	**{0,2}**	{1;0}	{1;0}	{1;0}	**{1;2}**
{1;2}	{1;2}	{1;2}	{1;2}	**{1;0}**	{1;2}	{1;2}

Table 2.2: Transition function $\Phi(\tilde{s}, e_p)$ for the model of Figure 2.2

[†]It is important to observe that the transition function Φ is a theoretical definition that is not necessarily used in current SAN solvers implementation, since the Kronecker descriptor present another structural perspective for the solution algorithms.

The SAN model construction as a Markov process has the rates of each event e_p seen as intensities α_p of Poisson processes (rates), and they are supposed to be independent. If an event e_p can lead to more than one state, it is needed to decompose the event e_p in new events as new Poisson processes with their respective rates. In the case of an event e_p with a functional rate, it is decomposed in as many events as possible evaluations i of the function. For each function evaluation we have a new Poisson process, and consequently a new event e_{p_i} associated.

The SAN description has now a table of events and their characteristics as basis. The idea of events uniformization is to introduce the independence among events applied successively. The uniformized process is driven by the Poisson process with rate $\alpha = \sum_{p=1}^{P} \alpha_p$ and generates at each time an event $e_p \in \xi$ according to the distribution $\left(\frac{\alpha_1}{\alpha}, \ldots, \frac{\alpha_p}{\alpha}\right)$.

Definition. The dynamic of the system is defined by one initial global state $\tilde{s}_0 \in \mathcal{X}$ and a sequence of events $e = \{e_p\}_{p \in \mathcal{N}}$. The sequence of states $\{\tilde{s}_n\}_{n \in \mathcal{N}}$ is a stochastic recursive sequence typically given by: $\tilde{s}_{n+1} = \Phi(\tilde{s}_n, e_{p+1})$ for $p \geq 0$ and is called a *trajectory*.

The global process execution [9, 60] described is related to the underlying uniformized Markov chain. Its transitions are given by Φ applications over $\mathcal{X}$. However, it is common to have global states that are not reachable by any other global state through a transition.

SAN models have $|\mathcal{X}^R|$ reachable states, so the others are considered unreachable global states in the model. Establishing the global state $\{0; 0\}$ as the initial global state $\tilde{s}_0$, we can have the reachable state space $\mathcal{X}^R_{\tilde{s}_0}$ (or $\mathcal{X}^R$) successively applying the defined transition functions. For our example, $\mathcal{X}^R \subseteq \mathcal{X}$ and the model is *well-formed* excluding the unreachable state $\{1; 1\}$. A simple procedure to find reachable states is to apply the notion of stochastic recursive transition function mainly when the reachability function is not explicit in the SAN formal descriptions[‡].

Considerations about events firing in SAN

One intrinsic characteristic of structured formalisms such as SAN is the multidimensional state space required. The global state of a system is established by the combination of local states of every component in the model. Considering a finite reachable state space $\mathcal{X}^R$ containing global states composed of automata local states, for each event in ξ, the global transition function $\Phi(\tilde{s}, e_i)$ application is given by an event e_i internally operating over each local state $s^{(k)}$ in $\tilde{s} = \{s^{(1)}; \ldots; s^{(K)}\}$, changing or not the global state.

There are two ways that an automaton can change its local state: with a synchronizing event (certainly affecting other automata) and with a local event. Both ways can be modeled with functional

[‡]SAN descriptions can define the $\mathcal{X}^R$ set through the insertion of a reachability function $\mathcal{F}^R$ (or a partial $\mathcal{F}^{R*}$). The boolean evaluation of this function, when applied to every global state inside $\mathcal{X}$, returns the reachable states in $\mathcal{X}^R$. More details can be found in [5, 49] and some explanations on Section 2.2.

dependencies based on different automata. An event activation must verify all involved automata, indicating that an event $e_p \in \xi$ is enabled (or disabled) considering each local state $s^{(i)} \in \delta^{(k)}$ of each automaton $A^{(k)}$ in the network.

Definition. The function $\varpi(e_p) = \{\mathcal{A}^{(i)}\}_{i \in \mathbb{N}}$ returns a set of automata directly affected by the event $e_p \in \xi$.

Definition. The local transition function $\phi(s^{(k)}, e_p)$ returns a new local state resulting of the event e_p firing over a local state $s^{(k)}$. The new local state returned can be the same $s^{(k)}$, if e_p does not affect it.

Considering a local state $s^{(k)}$ of automaton $\mathcal{A}^{(k)}$, which belongs to $\varpi(e_p)$. The ϕ function application results in the state $r^{(k)}$ means that the event is enabled for the local state $s^{(k)}$. If the ϕ function application results in the state $s^{(k)}$, consequently the event e_p is disabled for this local state.

Analogously, as mentioned in Section 2.3.2, given a global state $\tilde{s} = \{s^{(1)}; \ldots; s^{(K)}\}$, the global transition function can be viewed generically as $\Phi(\tilde{s}, e_p) = \tilde{r}$ if e_p is enabled for $\tilde{s}$, or $\Phi(\tilde{s}, e_p) = \tilde{s}$ if e_p is disabled for $\tilde{s}$.

Algorithm 2.1 Event firing verification procedure

1: $A_p \leftarrow \varpi(e_p)$ { A_p is the list of automata $\mathcal{A}^{(i)}$ involved in e_p}
2: { looking at each local state in $\tilde{s}$ }
3: **for all** $s^{(k)} \in \tilde{s} = \{s^{(1)}; \ldots; s^{(K)}\}$ **do**
4: { automaton $\mathcal{A}^{(k)}$ involved in event e_p }
5: **if** $\mathcal{A}^{(k)} \in A_p$ **then**
6: $r^{(k)} \leftarrow \phi(s^{(k)}, e_p)$
7: { the event e_p was not activated in the local state $s^{(k)}$ }
8: **if** $r^{(k)} = s^{(k)}$ **then**
9: { the global state $\tilde{s}$ did not change}
10: return $\Phi(\tilde{s}, e_p) = \tilde{s}$
11: **end if**
12: **end if**
13: **end for**
14: { the event was activated in all concerned automata, return next global state $\tilde{r}$ }
15: return $\Phi(\tilde{s}, e_p) = \tilde{r}$

Algorithm 2.1 shows the firing verification procedure, where each local state in $\tilde{s}$ related to an automaton included in $\varpi(e_p)$ is analyzed to fire an event e_p. Considering the example in the Figure 2.2, following we show some local transitions firings through ϕ, generating different global states.

Firing event e_2 from the global state $\tilde{s} = \{1^{(1)}; 0^{(2)}\}$:

$\varpi(e_2) = \{\mathcal{A}^{(1)}, \mathcal{A}^{(2)}\}$

$1^{(1)} \in \mathcal{A}^{(1)} \rightarrow \phi(1^{(1)}, e_2) = 0^{(1)}$

$0^{(2)} \in \mathcal{A}^{(2)} \rightarrow \phi(0^{(2)}, e_2) = 2^{(2)}$

New state: $\tilde{r} = \{0^{(1)}; 2^{(2)}\}$

Firing event e_4 from the global state $\tilde{s} = \{1^{(1)}; 2^{(2)}\}$:

$\varpi(e_4) = \{\mathcal{A}^{(2)}\}$

$2^{(2)} \in \mathcal{A}^{(2)} \rightarrow \phi(2^{(2)}, e_4) = 0^{(2)}$

New state: $\tilde{r} = \{1^{(1)}; 0^{(2)}\}$

Firing event e_3 from the global state $\tilde{s} = \{0^{(1)}; 0^{(2)}\}$:

$\varpi(e_3) = \{\mathcal{A}^{(2)}\}$

$0^{(2)} \in \mathcal{A}^{(2)} \rightarrow \phi(0^{(2)}, e_3) = 0^{(2)}$

New state: $\tilde{r} = \{0^{(1)}; 0^{(2)}\}$

Focusing on the structural aspects of SAN models, we look at the global states and the effect of events over them. Note that, for event firing purposes, it is not explicitly considered the event type, *i.e.*, all events present a synchronizing behavior even for the cases where it involves just one automaton. This means that the global state still changes independently of the events types defined in SAN descriptions.

This chapter summarized the SAN formalism background needed for the understanding of the solutions proposed in this thesis. Different SAN models descriptions are used as case studies for both numerical and theoretical results and they are presented in the Appendix A.

Chapter 3

Kronecker-based Descriptor Solution

The implementation of stationary and transient solvers must deal with a compact format through algorithms well-fitted to a multidimensional Kronecker representation of the infinitesimal generator $\mathcal{Q}$. This chapter discusses vector-descriptor product procedures for solution purposes. In practice, the structured (Kronecker) representation of $\mathcal{Q}$ can be obtained using different modeling formalisms beyond SAN, for example, stochastic Petri nets (SPN) [1] or even a less procedural approach but very modular description such as stochastic process algebra (PEPA) [44, 45]. The tensor principle [2, 26] recently has also been used in other stochastic formalisms [32, 45]. Thus, any structured formalism with a tensor representation, *e.g.*, SPN or PEPA, could employ the numerical algorithms of this chapter without any loss of generality.

The background needed to understand the numerical contribution of this thesis to SAN, involves concepts of classical tensor algebra (Appendix B) and specific notions of the SAN descriptor as an algebraic formula (Sections 2.1 and 2.3.1).

3.1 Vector-Descriptor Product

Assuming that the underlying Markov chain is irreducible and it does not contain unreachable states, for many applications it can be large and composed of many nonzero elements. Due to this, the compact representation is a valid alternative, enabling even larger systems to be described and solved. In order to efficiently analyze Markovian models based on Kronecker products, three algorithms for vector-descriptor product are proposed [6, 16, 37] and implemented as the core for iterative solution techniques in different modeling formalisms. These formalisms are integrated in different software packages such as SMART (Stochastic Model Checking Analyzer for Reliability and Timing) [22], PEPS (Performance Evaluation of Parallel Systems) [5, 49] and the PEPA Workbench [41]. Concerning the descriptor structure, there are solution approaches proposed, varying from hy-

brid solutions for simulation and numerical analysis [15] to algorithms adapted to alternative storage structures [16, 52].

Despite the algorithmic differences, the approaches can be summarized in finding an efficient way to multiply a (usually huge) vector by a non-trivial structure (a *descriptor*) inside an iterative method, *e.g.*, *Power* method [59, 61]. The aim is to obtain the stationary distribution π related to the model. Old stochastic Petri net solutions [1] translate the model representation into a singular sparse matrix. Obviously, this sparse approach is difficult to be employed for really large models (*e.g.*more than 500 thousand states), since it usually requires the storage of a too large sparse matrix (*e.g.*more than 4 million nonzero elements). This is only possible with non-trivial solutions such as disk-based approaches [27], *On-the-fly* generation techniques [28] or even parallel implementations [4, 31, 62].

The usual SAN solution is the *Shuffle* algorithm [26, 37] and it deals with permutations of matrices and tensor products. However, in an algebraic view, this algorithm can be applied to the analysis of any Markov chain based on Kronecker products, independently from the modeling formalism. The vector-descriptor multiplication, in this case, corresponds to the product of a probability vector υ, as big as the model product state space, by a descriptor $\mathcal{Q}$ in a Kronecker representation. Therefore, by a simple distributive property, vector-descriptor product algorithms can be viewed in a simpler format as a sum of $K + 2E$ products of the a vector υ by a tensor product term composed by K matrices (Equation 3.1):

$$\sum_{j=1}^{K+2E} \left(\upsilon \times \left[\bigotimes_{k=1}^{K} \mathcal{Q}_j^{(k)} \right] \right) \tag{3.1}$$

The research related to the numerical solutions of structured huge Markov models aims to speedup the basic operation $\upsilon \times \left[\bigotimes_{k=1}^{K} \mathcal{Q}_j^{(k)} \right]$. In this section, classical algorithms used to compute the multiplication of a vector by Markovian descriptors are presented, showing their advantages and limitations when tensor structures are used.

3.1.1 *Sparse* solution

The sparse solution is the most intuitive method to solve the mapping of the Kronecker structure into a matrix containing only nonzero elements multiplied by a probability vector. The numerical algorithm is called in this thesis the *Sparse* algorithm. It aims to consider a tensor product term $\mathcal{T}$ explicitly as a single sparse matrix, multiplying it by a product state space sized probability vector. This means that an evaluation of the algebraic expression leads to a matrix of size equal to the product state space.

Considering a tensor product $\mathcal{T}$ of K matrices $\mathcal{Q}^{(k)}$, each one of dimension n_k, and with nz_k nonzero elements, the *Sparse* algorithm generates element by element of one large matrix $\mathcal{Q} =$

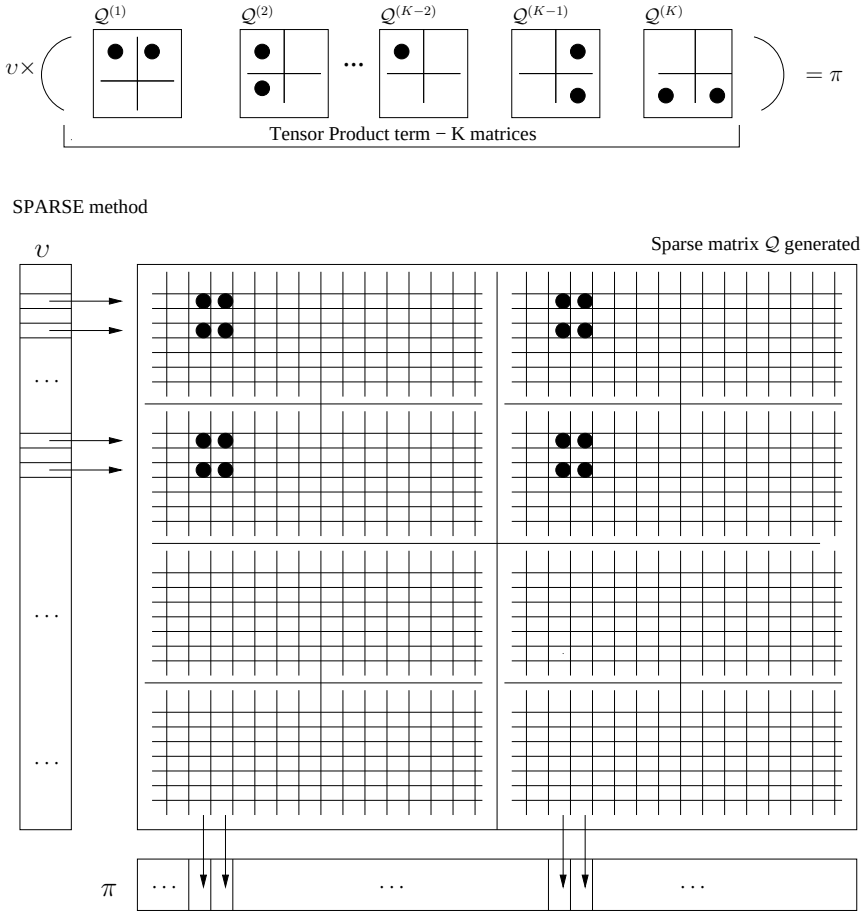


Figure 3.1: *Sparse* method illustration

$\otimes_{k=1}^{K}\mathcal{Q}^{(k)}$, with order $\prod_{k=1}^{K} n_k$. Then, the corresponding elements of vector υ (with dimension given by $\prod_{k=1}^{K} n_k$) are multiplied by the sparse matrix $\mathcal{Q}$, storing the results in the probability vector π. Figure 3.1 shows an illustration of the *Sparse* method where the nonzero elements in the matrix $\mathcal{Q}$ are represented generically by the small black circles in the matrices.

This approach has storage disadvantages but the time-efficiency is less intuitive, because the full matrix storage is not necessary if one can generate each nonzero of $\mathcal{Q}$ as fast as it is needed in the multiplication. According to the classical tensor product definition (refer to Appendix B) this operation requires $K-1$ additional multiplications to generate a nonzero element a.

Defining $\theta_{(1\ldots K)}$ as the set of all possible combinations of nonzero elements of the matrices from $\mathcal{Q}^{(1)}$ to $\mathcal{Q}^{(K)}$, the cardinality of $\theta_{(1\ldots K)}$, *i.e.*, the number of nonzero elements in $\mathcal{Q}$ is given by: $\prod_{k=1}^{K} nz_k$. Additionally, the *Sparse* method needs the information of the dimension of the state space corresponding to all matrices after the k^{th} matrix of the tensor product, called $nright_k$ (numerically defined by $\prod_{i=k+1}^{K} n_i$). We also have the analogous concept of $nleft_k$ which is numerically equal to

Algorithm 3.1 *Sparse* algorithm - $\pi = \upsilon \times \otimes_{k=1}^{K} \mathcal{Q}^{(k)}$

1: $\Upsilon = 0$
2: **for all** $i_1, \ldots, i_K, j_1, \ldots, j_K \in \theta(1 \ldots K)$ **do**
3: $\quad a = 1$
4: $\quad base_{in} = base_{out} = 0$
5: $\quad$ **for all** $k = 1, 2, \ldots, K$ **do**
6: $\quad\quad a = a \times q^{(k)}_{(i_k, j_k)}$
7: $\quad\quad base_{in} = base_{in} + ((i_k - 1) \times \ nright_k)$
8: $\quad\quad base_{out} = base_{out} + ((j_k - 1) \times \ nright_k)$
9: $\quad$ **end for**
10: $\quad \pi[base_{out}] = \pi[base_{out}] + \upsilon[base_{in}] \times a$
11: **end for**

$\prod_{i=1}^{k-1} n_i$. The *Sparse* method is presented in the Algorithm 3.1.

The *Sparse* computational cost considering the number of floating point multiplications is given by (Equation 3.2):

$$K \times \prod_{k=1}^{K} nz_k \tag{3.2}$$

However, in this algorithm, all nonzero elements of $\mathcal{Q}$ are generated during the algorithm execution. Such elements generation represents $(K-1) \times \prod_{k=1}^{K} nz_k$ multiplications that could be avoided by generating one (usually huge) sparse matrix to store these $\prod_{k=1}^{K} nz_k$ nonzero elements. It would eliminate the lines 3 and 6 from the *Sparse* algorithm and reduce the number of floating point multiplications to just (Equation 3.3):

$$\prod_{k=1}^{K} nz_k \tag{3.3}$$

This option allows the *Sparse* algorithm to be very time-efficient compared to specialized algorithms for the treatment of Kronecker products, but potentially very memory demanding due to the storage of a, potentially huge, sparse matrix $\mathcal{Q}$. Another interesting approach to the sparse algorithm is to keep the nonzero elements generation inside the algorithm, but factorizing previous calculations [16]. Note that all combinations of elements, of each matrix of the tensor product, have multiplications in common, *i.e.*, the nonzero elements can be generated considering partial multiplication results. Such solution can reduce the complexity in terms of number of multiplications applying an algorithm to exploit levels of factoring, reusing previous calculations.

In the context of this work, the sparse approach to be considered will be the time-efficient version, *i.e.*, the variant with the previous generation and storage of nonzero elements of $\mathcal{Q}$. Such variant demands a smaller number of floating point multiplications (Equation 3.3) than the traditional specialized algorithm for tensor products, but it stores a sparse matrix with $\prod_{k=1}^{K} nz_k$ nonzero elements.

3.1.2 The memory-efficient *Shuffle* algorithm

A structured view of infinitesimal generators represented by Kronecker algebra leads us to a specialized algorithm which deals with building blocks of nonzero elements, performing shuffling operations in the probability vector υ. The immediate effect of using tensor properties to optimize the numerical solution is the reduction of the memory spent, moving this bottleneck from the infinitesimal generator to the probability vector.

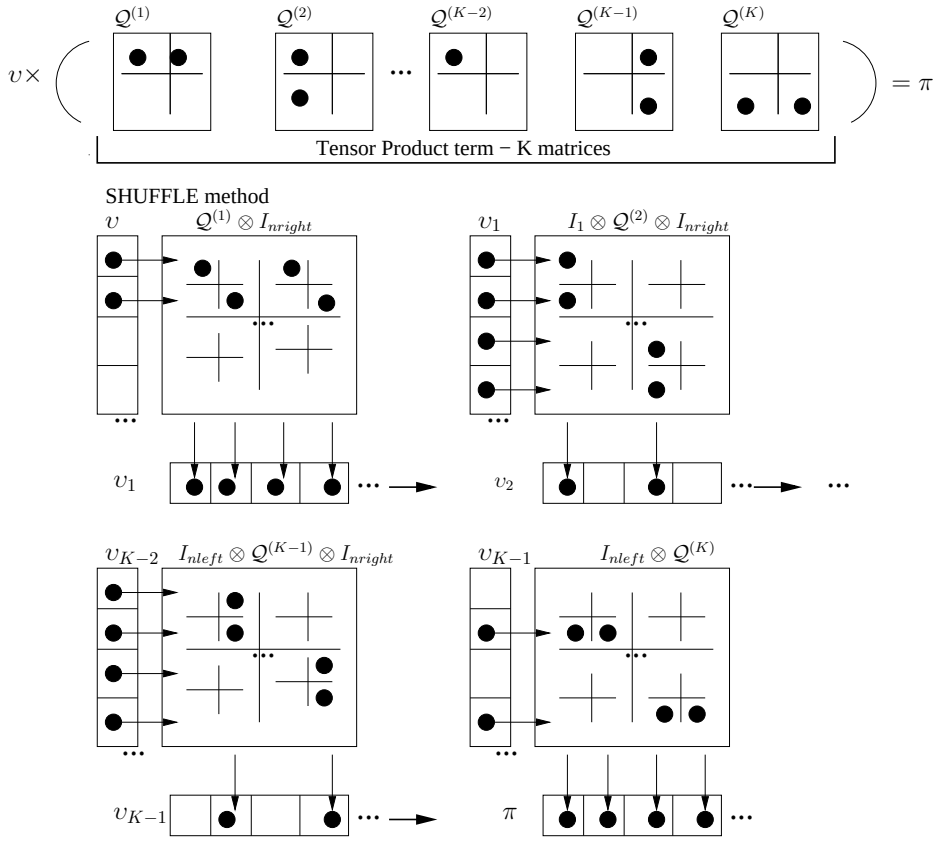


Figure 3.2: *Shuffle* method illustration

The basic principle of the *Shuffle* algorithm is the application of the decomposition of a tensor product $\mathcal{T}$ in the ordinary product of normal factors property [37] (Equation 3.4):

$$\begin{array}{lcl} \mathcal{T} = \mathcal{Q}^{(1)} \otimes \mathcal{Q}^{(2)} \otimes \ldots \otimes \mathcal{Q}^{(K-1)} \otimes \mathcal{Q}^{(K)} & = & \\ (\mathcal{Q}^{(1)} \otimes I_{n_2} \otimes \ldots \otimes I_{n_{K-1}} \otimes I_{n_K}) & \times & (I_{n_1} \otimes \mathcal{Q}^{(2)} \otimes \ldots \otimes I_{n_{K-1}} \otimes I_{n_K}) \quad \times \\ \ldots & \times & \ldots \quad \times \\ (I_{n_1} \otimes I_{n_2} \otimes \ldots \otimes \mathcal{Q}^{(K-1)} \otimes I_{n_K}) & \times & (I_{n_1} \otimes I_{n_2} \otimes \ldots \otimes I_{n_{K-1}} \otimes \mathcal{Q}^{(K)}) \end{array} \tag{3.4}$$

Hence, the *Shuffle* algorithm consists in multiplying successively a vector υ by each tensor term decomposed in normal factors. More precisely, υ is multiplied by the first normal factor $\mathcal{Q}^{(1)} \otimes I_{nright_1}$, then the resulting vector is multiplied by the next one $I_{nleft_k} \otimes \mathcal{Q}^{(k)} \otimes I_{nright_k}$, and so on, until the last multiplication by $I_{nleft_K} \otimes \mathcal{Q}^{(K)}$.

Internally to each normal factor, the multiplications are done using small auxiliary vectors called z_{in} and z_{out} in the Algorithm 3.2. These small vectors store the values of υ to multiply by the k^{th} matrix of the normal factor, and then store the result in the probability vector π (Figure 3.2). The vectors dimensions are given by the matrix $\mathcal{Q}^{(k)}$ dimension in each normal factor multiplied.

Algorithm 3.2 *Shuffle* algorithm - $\pi = \upsilon \times \otimes_{k=1}^{K} \mathcal{Q}^{(k)}$

```
1: for all k = 1, 2, ..., K do
2:    base = 0
3:    for all m = 0, 1, 2, ..., nleft_k − 1 do
4:       for all j = 0, 1, 2, ..., nright_k − 1 do
5:          index = base + j
6:          for all l = 0, 1, 2, ..., n_k − 1 do
7:             z_in[l] = υ[index]
8:             index = index + nright_k
9:          end for
10:         multiply  z_out = z_in × Q^(k)
11:         index = base + j
12:         for all l = 0, 1, 2, ..., n_k − 1 do
13:            υ[index] = z_out[l]
14:            index = index + nright_k
15:         end for
16:      end for
17:      base = base + ( nright_k × n_k)
18:   end for
19: end for
20: π = υ
```

Generalizing the multiplication of a vector υ by the k^{th} normal factor, it consists in *shuffling* the elements of υ in order to assemble $nleft_k \times nright_k$ vectors of size n_k, multiplying them by matrix $\mathcal{Q}^{(k)}$. Thus, assuming that matrix $\mathcal{Q}^{(k)}$ is stored as a sparse matrix, the number of operations needed to multiply a vector by the k^{th} normal factor is: $nleft_k \times nright_k \times nz_k$, where nz_k corresponds to the number of nonzero elements of the k^{th} matrix $\mathcal{Q}^{(k)}$ of the tensor product term.

Considering the number of multiplications of all normal factors in a tensor product term, the *Shuffle* computational cost to perform the basic operation (multiplication of a vector by $\mathcal{T}$) is given by [37] (Equation 3.5):

$$\sum_{k=1}^{K} nleft_k \times nright_k \times nz_k = \prod_{k=1}^{K} n_k \times \sum_{k=1}^{K} \frac{nz_k}{n_k} \tag{3.5}$$

Another feature of the *Shuffle* algorithm is the product optimization for functional elements, *i.e.*, the use of generalized tensor algebra properties and matrices reordering [37]. All those optimizations are very important to reduce the overhead of evaluating functional elements, but such considerations are out of the scope of this thesis. Our focus is the vector-descriptor considering the tensor terms are described with classical tensor algebra (more details about its properties are found in the Appendix B).

3.2 The Hybrid *Split* Algorithm

SAN models of practical applications are often sparse. Moreover, the tensor sum in a descriptor is intrinsically very sparse due to the normal factors structure. The dependent behaviors represented by tensor products let this part of the descriptor also quite sparse. These characteristics lead us to propose a hybrid approach called *Split* algorithm [24] that takes advantage of matrices sparsity combined with the advantages of the classical tensor product decomposition in normal factors.

The additive decomposition property, applied to any tensor product term $\mathcal{T}$, states that a term can be decomposed into an ordinary sum of matrices (composed by one single nonzero element). Noe that it is also the principle of sparse techniques.

Assuming $\hat{q}_{(i_1,\ldots,i_{K-1},j_1,\ldots,j_K)}$ the matrix of dimension $\prod_{i=1}^{K} n_i$ composed by only one nonzero element which is in position $i_1, \ldots, i_K, j_1, \ldots, j_K$ and it is equal to $\prod_{k=1}^{K} q_{i_k,j_k}^{(k)}$, the additive decomposition property is given by (Equation 3.6):

$$\begin{aligned} \mathcal{T} = \mathcal{Q}^{(1)} \otimes \mathcal{Q}^{(2)} \otimes \ldots \otimes \mathcal{Q}^{(K-1)} \otimes \mathcal{Q}^{(K)} = \\ \sum_{i_1=1}^{n_1} \cdots \sum_{i_K=1}^{n_K} \sum_{j_1=1}^{n_1} \cdots \sum_{j_K=1}^{n_K} \left(\hat{q}_{(i_1,j_1)}^{(1)} \otimes \ldots \otimes \hat{q}_{(i_K,j_K)}^{(K)} \right) \end{aligned} \tag{3.6}$$

where $\hat{q}_{(i,j)}^{(k)}$ is a matrix of order n_k in which the element in row i and column j is $q_{i,j}^{(k)}$.

The *Split* method proposes a combined solution using an additive property for a given set of matrices inside a tensor product term $\mathcal{T}$, performing the shuffling operations for the other matrices. Hence, each tensor product of matrices can be partitioned (or splitted) in two different groups: the first one with the most sparse matrices; and the second one with the matrices with a larger number of nonzero elements. This is possible when permutations* of matrices are allowed.

A *Sparse*-like approach could be applied to the first group of K matrices generating new factors called *Additive Unitary Normal Factors* (*AUNF*s). An *AUNF* presents a scalar value a associated and

*It is out of the scope of this thesis to analyze permutations and the effect of them for the *Split* algorithm.

Split σ	Tensor Product Term $\mathcal{T} = \otimes_{i=1}^{K} \mathcal{Q}^{(i)}$
0	$\overset{\sigma}{\downarrow}$ $\mathcal{Q}^{(1)} \otimes \mathcal{Q}^{(2)} \otimes \ldots \otimes \mathcal{Q}^{(K-3)} \otimes \mathcal{Q}^{(K-2)} \otimes \mathcal{Q}^{(K-1)} \otimes \mathcal{Q}^{(K)}$ — *Shuffle*
1	$\overset{\sigma}{\downarrow}$ $\mathcal{Q}^{(1)} \otimes \mathcal{Q}^{(2)} \otimes \ldots \otimes \mathcal{Q}^{(K-3)} \otimes \mathcal{Q}^{(K-2)} \otimes \mathcal{Q}^{(K-1)} \otimes \mathcal{Q}^{(K)}$ — *Sparse*; *Shuffle*
2	$\overset{\sigma}{\downarrow}$ $\mathcal{Q}^{(1)} \otimes \mathcal{Q}^{(2)} \otimes \ldots \otimes \mathcal{Q}^{(K-3)} \otimes \mathcal{Q}^{(K-2)} \otimes \mathcal{Q}^{(K-1)} \otimes \mathcal{Q}^{(K)}$ — *Sparse*; *Shuffle*
⋮	⋮
K-2	$\overset{\sigma}{\downarrow}$ $\mathcal{Q}^{(1)} \otimes \mathcal{Q}^{(2)} \otimes \ldots \otimes \mathcal{Q}^{(K-3)} \otimes \mathcal{Q}^{(K-2)} \otimes \mathcal{Q}^{(K-1)} \otimes \mathcal{Q}^{(K)}$ — *Sparse*; *Shuffle*
K-1	$\overset{\sigma}{\downarrow}$ $\mathcal{Q}^{(1)} \otimes \mathcal{Q}^{(2)} \otimes \ldots \otimes \mathcal{Q}^{(K-3)} \otimes \mathcal{Q}^{(K-2)} \otimes \mathcal{Q}^{(K-1)} \otimes \mathcal{Q}^{(K)}$ — *Sparse*; *Shuffle*
K	$\overset{\sigma}{\downarrow}$ $\mathcal{Q}^{(1)} \otimes \mathcal{Q}^{(2)} \otimes \ldots \otimes \mathcal{Q}^{(K-3)} \otimes \mathcal{Q}^{(K-2)} \otimes \mathcal{Q}^{(K-1)} \otimes \mathcal{Q}^{(K)}$ — *Sparse*

Table 3.1: *Split* as a generalization of traditional algorithms

indexes (line and column) related to the combination of the nonzero elements of each matrix in the group. The scalar a is the result of this combination (or matrices *aggregation*) computed using the nonzero values found in the matrices.

Each one of those *AUNF*s should be tensorly multiplied by the second group of matrices using a *Shuffle*-like approach. The *Shuffle* algorithm is applied normally in this subset of matrices considering that the probability vector υ to be multiplied, has already the information related to the *AUNF*. The shuffling operation is applied to each *AUNF* generated from the first group of matrices. The idea is to *split* the tensor terms in two sets of matrices treating them in two different ways. Table 3.1 presents the general idea of *Split* graphically. The index of the matrix chosen for delimiting the end of the first set of matrices is called *cut-parameter* σ. It is possible to observe that the *Sparse* ($\sigma = K$) and the *Shuffle* ($\sigma = 0$) methods are particular cases of the *Split* algorithm.

The Figure 3.3 shows a tensor product term of K matrices indicating a *cut-parameter* σ after the $\mathcal{Q}^{(K-2)}$ matrix. The sparse part will generate the combinations of nonzero elements of each matrix in this subset, calculating the *AUNF*s. Supposing we have three matrices in this part with $nz_1 = 2$, $nz_2 = 2$ and $nz_3 = 1$ respectively (nz_i is the number of nonzero elements of each matrix). Consequently, the combination of these elements generates four *AUNF*s ($\prod_{i=1}^{\sigma} nz_i = 2 \times 2 \times 1$) and

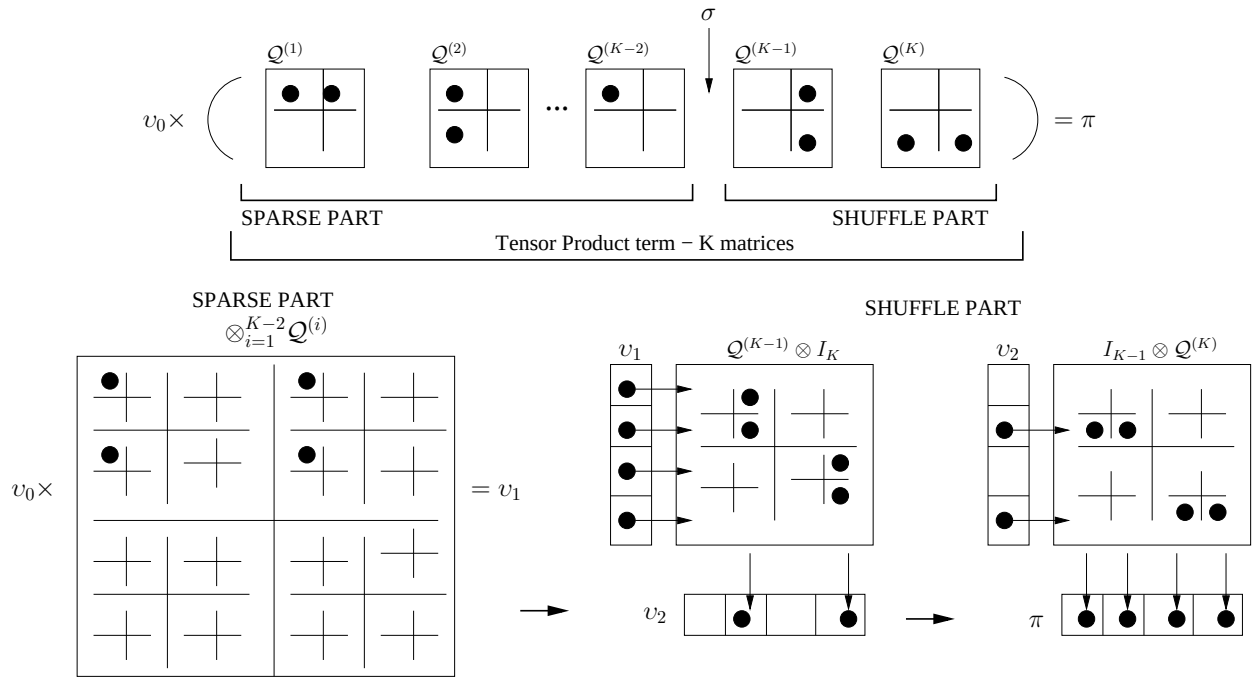


Figure 3.3: *Split* method illustration

their indexes in the implicit matrix (correspondent to the sparse part) are calculated based on the lines and columns of each nonzero element considered. The shuffle part indicated at right in the Figure 3.3 is applied to each *AUNF* of the sparse part individually. This means that the first *AUNF* is multiplied by the vector v_0, then the resultant vector v_1 is used as input to the shuffle part, accumulating the result on the vector π. After that, the second *AUNF* is multiplied by the vector v_0, and so on, until the last scalar in the sparse part.

Algorithm 3.3 defines formally the steps of the splitting procedure using the notation v for the input vector and v_{in} the auxiliary vectors in the multiplication. It consists in the computation of the scalar element a of each *AUNF* in $\theta(1 \ldots \sigma)$ by multiplying one nonzero element of each matrix of the first set of matrices from $\mathcal{Q}^{(1)}$ to $\mathcal{Q}^{(\sigma)}$ (lines 2 to 9). According to the elements row indexes used to generate the scalar element a, a contiguous slice of input vector v, called v_{in}, is taken. Vector v_{in} of size $nright_\sigma$ (corresponding to the product of the order of all matrices after the *cut-parameter* σ of the tensor product term) is multiplied by the element a. Lines 10 to 11 perform the multiplication of the scalar element composing the *AUNF* for each position in v, then finishing the sparse part.

The resulting vector also v_{in} is used as input vector to the *Shuffle*-like multiplication (lines 13 to 31) by the tensor product of the matrices in the second set of matrices (from $\mathcal{Q}^{(\sigma+1)}$ to $\mathcal{Q}^{(K)}$). At the end of the Shuffle part, the vector v obtained is accumulated in the final vector π (lines 32 to 34). Note that the *cut-parameter* σ is pre-defined before running the algorithm, which means one may define

Algorithm 3.3 *Split* algorithm - $\pi = \upsilon \times \otimes_{i=1}^{K} \mathcal{Q}^{(k)}$

$\Upsilon = 0$
for all $i_1, \ldots, i_\sigma, j_1, \ldots, j_\sigma \in \theta(1 \ldots \sigma)$ **do**
  $a = 1$
  $base_{in} = base_{out} = 0$
  for all $k = 1, 2, \ldots, \sigma$ **do**
    $a = a \times q^{(k)}_{(i_k, j_k)}$
    $base_{in} = base_{in} + ((i_k - 1) \times nright_k)$
    $base_{out} = base_{out} + ((j_k - 1) \times nright_k)$
  end for
  for all $l = 0, 1, 2, \ldots, nright_\sigma - 1$ **do**
    $\upsilon_{in}[l] = \upsilon[base_{in} + l] \times a$
  end for
  for all $i = \sigma + 1, \ldots, K$ **do**
    $base = 0$
    for all $m = 0, 1, 2, \ldots, \frac{nleft_i}{nleft_\sigma} - 1$ **do**
      for all $j = 0, 1, 2, \ldots, nright_i$ **do**
        $index = base + j$
        for all $l = 0, 1, 2, \ldots, n_i - 1$ **do**
          $z_{in}[l] = \upsilon_{in}[index]$
          $index = index + nright_i$
        end for
        $multiply \;\; z_{out} = z_{in} \times \mathcal{Q}^{(i)}$
        $index = base + j$
        for all $l = 0, 1, 2, \ldots, n_i - 1$ **do**
          $\upsilon_{in}[index] = z_{out}[l]$
          $index = index + nright_i$
        end for
      end for
      $base = base + (nright_i \times n_i)$
    end for
  end for
  for all $l = 0, 1, 2, \ldots, nright_\sigma - 1$ **do**
    $\pi[base_{out} + l] = \pi[base_{out} + l] + \upsilon_{in}[l]$
  end for
end for

this division point considering the memory available to store and manipulate the *AUNF*s as well as use the characteristics of the tensor product terms formations to decide the best *cut-parameter* in each situation.

The hybrid solution, depending on the sparsity of descriptors, can do better than *Shuffle* algorithm since most matrices can have few nonzero elements which has an impact in the number of *AUNF*s. *Shuffle* pays an additional overhead in time for the efficient saving in space because it deals with complex indexes calculations. The next section shows the computational costs of the *Split* algorithm considering also some optimizations. Following, we show the numerical results of the *Split* algorithm application in SAN descriptors, presenting an algorithm for sampling a well-fitted *cut-parameter* σ for each tensor product term.

3.2.1 Theoretical contributions

This section presents the computational cost in number of multiplications (Equation 3.7) for the *Split* algorithm, which is also considered a theoretical contribution in thesis, since the operations involved are reduced when compared to the *Shuffle* algorithm computational cost (Equation 3.5).

The computational cost is calculated taking into account the number of multiplications performed to generate each scalar element composing an *AUNF* $(\sigma - 1$ multiplications$)$, plus the number of multiplications of the scalar e by each position value of the vector v_{in}. There is also the cost to multiply the values in the input vector v_{in} by the tensor product of matrices in the *Shuffle*-like part.

$$\left(\prod_{i=1}^{\sigma} nz_i\right)\left[(\sigma-1)+\left(\prod_{i=\sigma+1}^{K} n_i\right)+\left(\prod_{i=\sigma+1}^{K} n_i \times \sum_{i=\sigma+1}^{K} \frac{nz_i}{n_i}\right)\right] \tag{3.7}$$

In practical implementations of vector-descriptor multiplication algorithms, improvements can also be done to speedup the execution. These optimizations can change significantly the theoretical computational cost presented in the Equation 3.7.

Regarding the *Shuffle* method, there is an optimization on the way of handling identity matrices. Those matrices do not need to generate normal factors, since being identity matrices, they generate a normal factor that is also an (huge) identity matrix itself. The computational cost is clearly reduced in the *Shuffle* algorithm execution when using this solution. It corresponds to transform the number of floating point multiplications equation for the *Shuffle* algorithm (Equation 3.5) to Equation 3.8:

$$\prod_{i=1}^{K} n_i \times \sum_{\substack{i=1 \\ iff Q^{(i)} \neq Id}}^{K} \frac{nz_i}{n_i} \tag{3.8}$$

This improvement suggests the same skipping-identities optimization to the *Shuffle*-like part (ma-

trices $\mathcal{Q}^{(\sigma+1)}$ to $\mathcal{Q}^{(K)}$) of the *Split* (Algorithm 3.3) identifying if the matrix indexed by variable i of the algorithm ($\mathcal{Q}^{(i)}$) is not an identity matrix, adding to the cost $\frac{nz_i}{n_i}$ multiplications only for these ones. Analogously to *Shuffle* algorithm, Equation 3.7 will be rewritten changing the *Shuffle*-part cost accordingly to σ. The resulting number of floating point multiplications for the *Split* algorithm will be (Equation 3.9):

$$\left(\prod_{i=1}^{\sigma} nz_i\right)\left[(\sigma-1)+\left(\prod_{i=\sigma+1}^{K} n_i\right)+\left(\prod_{i=\sigma+1}^{K} n_i \times \sum_{\substack{i=\sigma+1 \\ iff\mathcal{Q}^{(i)}\neq Id}}^{K} \frac{nz_i}{n_i}\right)\right] \tag{3.9}$$

Usually the tensor product terms of a SAN model are very sparse (a few thousands nonzero elements). The only cases where a more significant number of nonzero elements are found, are those when we are dealing with a tensor product term with many identity matrices. It is important to recall that to each *AUNF*, the scalar a is computed as the product of one single element of each matrix.

The second optimization is the precomputation of these nonzero elements and their storage. This optimization was already largely studied in [16]. It results consequently in a reduction of the *Split* algorithm computational cost, similar to that one presented in Section 3.1.1 (Equation 3.2) regarding the computation of nonzero elements. Hence, the final definition of the number of floating point multiplications for the *Split* method is no longer defined by Equation 3.9, but as (Equation 3.10):

$$\left(\prod_{i=1}^{\sigma} nz_i\right)\left[\left(\prod_{i=\sigma+1}^{K} n_i\right)+\left(\prod_{i=\sigma+1}^{K} n_i \times \sum_{\substack{i=\sigma+1 \\ iffQ^{(i)}\neq Id}}^{K} \frac{nz_i}{n_i}\right)\right] \tag{3.10}$$

For very sparse tensor products the best cut-parameter σ points to a pure sparse approach mainly because few *AUNF* will be generated avoiding the shuffling operations. The method, obviously, is much more effective if the nonzero elements do not have to be recomputed at each vector multiplication. Therefore, the *Split* algorithm must balance together the computational cost in terms of multiplications and its memory needs considering the *descriptor* plus the size of the new structures such as the *AUNF*. The next section presents the practical results obtained, using the equations above to demonstrate the gains achieved.

3.2.2 Practical contributions

The collection of classical and practical [3, 34] examples (Appendix A) counts with a variety of tensor product formations, since semantic aspects allow different automata interconnections through more or less synchronizing (or local) events. Despite the significant computational cost re-

duction given by functional transitions, functions are not mandatory when modeling or solving with SAN. Models with functions are converted to an equivalent representation† using only synchronizing events [13].

The performance results were collected running the algorithm implementation, varying the *cut-parameter* σ from *Shuffle* to *Sparse*, on a 3.2 GHz Intel Xeon under Linux operating system with 4 Gb of memory. The prototype module is inside the *PEPS2003* environment and was compiled using the `g++` compiler with optimizations ($-O3$).

All tensor product terms $\mathcal{T}$ of each Kronecker descriptor $\mathcal{Q}$ were executed in all possible *cut-parameters* σ, collecting time outputs for 100 runs (samples). The results were obtained in time intervals with 95% of confidence. The *Split* algorithm executions times presented in the tables are the sum of the average best execution times obtained for each tensor product term composing $\mathcal{Q}$, considering their different cut-parameters σ and a ν number of samples, according to the Algorithm 3.4.

Algorithm 3.4 Tensor terms execution times for a σ sampling

1: **for all** $\mathcal{T} \in \mathcal{Q}$ **do**
2: **for all** $\sigma \in N$ **do**
3: Run ν samples collecting execution times t_σ;
4: **end for**
5: **end for**
6: Calculate the confidence interval for the execution times t_σ;
7: Identify the fastest execution time t_σ within the confidence interval;
8: **for all** $\sigma \in N$ **do**
9: take execution time t_σ;
10: verify *if* $t_\sigma > t$ *then* discard t_σ and go to next σ;
11: **end for**
12: **for all** t_σ non-discarded **do**
13: mark t_σ with the less memory needed;
14: **end for**
15: **for all** $\mathcal{T} \in \mathcal{Q}$ **do**
16: assign the marked t_σ to $\sigma_\mathcal{T}$;
17: **end for**

Each tensor term $\mathcal{T}$ received one *cut-parameter* $\sigma_\mathcal{T}$ after the execution of Algorithm 3.4. When $\sigma_\mathcal{T}$ requires an unstructured part in the descriptor the procedure of AUNFS generation is activated to store the elements and their indexes. All other iterations in the method needed to solve the model descriptor are executed using $\sigma_\mathcal{T}$ and data structures related to this choice.

All tables in this section show the results obtained for the three methods: *Shuffle*, *Split* and *Sparse*, analyzing each one in three columns, representing the time spent in seconds per iteration (*sec.*), size

†The impact analysis of this translation to obtain the numerical solution is out of the scope of this work.

in *Kb* and the computational cost in floating point multiplications (*fpm*) following the equation 3.10. The column named $\mathcal{X}$ stands for the product state space (the dimension of the probability vector).

We also preserve the predefined order of automata given by the model description, do not performing automata permutations[‡] before running *Split*. In such way, it is possible to say that the *Split* results presented here tend to force the trade-off between time and memory efficiency towards time savings. The opposite choice (memory savings) would force the *Split* method to obtain practically the same efficiency of the *Shuffle* algorithm, known as a memory-efficient solution.

Resource Sharing model results

This section evaluates the *Resource Sharing* SAN model results (see Figure A.2). Table 3.2 shows different network configurations named as *P_R*, indicating the number P of processes and R of resources. Note that the *fpm* columns remains the same for *Split* and *Sparse* columns, so due to tensor terms sparsity this cost is the same for both approaches and better than the pure application of *Shuffle*.

Models (*P_R*)	$\mathcal{X}$	*Shuffle*			*Split*			*Sparse*		
		time (s)	size (Kb)	fpm	time (s)	size (Kb)	fpm	time (s)	size (Kb)	fpm
10_16	17,408	0.04851	**11.25**	1,003,520	**0.01550**	2,018.25	327,680	0.01636	5,131.25	**327,680**
10_20	21,504	0.06211	**13.75**	1,249,280	**0.01910**	2,373.75	409,600	0.02009	6,413.75	**409,600**
11_11	24,576	0.07477	**8.94**	1,531,904	**0.02304**	2,615.42	495,616	0.02448	7,752.94	**495,616**
11_14	30,720	0.09387	**11.00**	1,937,408	**0.02906**	3,524.13	630,784	0.03187	9,867.00	**630,784**
12_12	53,248	0.17808	**10.50**	3,637,248	**0.05793**	5,674.31	1,179,648	0.06897	18,442.50	**1,179,648**
13_13	114,688	0.42021	**12.19**	8,519,680	**0.15609**	10,859.88	2,768,896	0.20394	43,276.19	**2,768,896**
14_10	180,224	0.72949	**10.50**	14,221,312	**0.27686**	16,654.25	4,587,520	0.34378	71,690.50	**4,587,520**
14_11	196,608	0.80280	**11.38**	15,597,568	**0.30455**	18,316.41	5,046,272	0.37499	78,859.38	**5,046,272**

Table 3.2: *Resource Sharing* SAN model results

The maximum resource needed in memory, in these examples, is achieved by the last model with almost $\sim$19 Mb used. The reduction of the time spent per iteration compared with the *Shuffle* results shows the improvement obtained using the flexibility of the *Split* algorithm. The *Split* method is two times faster solving this models against the *Shuffle* which is five times less memory consuming.

Dining Philosophers model (with resource reservation)

This section presents the *Dining Philosophers* SAN model results (Figure A.4). Table 3.3 shows the results for K philosophers in the network. The model analyzed supposes a resource reservation

[‡]The *Shuffle* algorithm implements automata permutations to optimize the solution of descriptors mainly those described with generalized tensor algebra [37].

by the philosophers meaning each automata $Ph^{(k)}$ has three states and consequently the matrices composing the descriptor have a fixed dimension $n_k = 3$.

Models (K)	$\mathcal{X}$	*Shuffle*			*Split*			*Sparse*		
		time (s)	size (Kb)	fpm	time (s)	size (Kb)	fpm	time (s)	size (Kb)	fpm
6	729	0.00116	**1.69**	17,496	**0.00058**	17.09	13,365	0.00042	92.81	**5,832**
7	2,187	0.00421	**1.97**	61,236	**0.00241**	47.52	50,787	0.00153	320.91	**20,412**
8	6,561	0.01447	**2.25**	209,952	**0.00477**	242.84	76,545	0.00537	1,095.75	**69,984**
9	19,683	0.04639	**2.53**	708,588	**0.01632**	1,106.91	236,196	0.01740	3,693.09	**236,196**
10	59,049	0.15348	**2.81**	2,361,960	**0.05753**	4,035.28	787,320	0.06492	12,304.69	**787,320**
11	177,147	0.53765	**3.09**	7,794,468	**0.25230**	7,387.28	2,598,156	0.29316	40,599.28	**2,598,156**

Table 3.3: *Dining Philosophers* SAN model results

Note that the *fpm* needed in each method does not mean that a method will be faster in seconds, it is also related also to the access of structures in memory according to the *cut-parameters* σ of each tensor term. A structured descriptor exploitation forces the algorithm to access more positions in memory to update indexes and store results. However it is clear that this is dependable of tensor term formation, *i.e.* the number of non-zero elements in each matrix and the number of identity matrices in the term define the number of scalars in the sparse part and the dimension of the structured part.

The *Split* algorithm presents a time efficiency superior than the *Sparse* method and it means that sometimes structured behavior obtained with more shuffling operations can bring better performance not only regarding memory savings as expected. For the model with $K = 11$ is spent $\sim$7 Mb of extra memory to obtain the *Split* performance as close as possible to the *Sparse* method computational time.

First Available Server model results

This section presents the results for the *First Available Server* SAN model (Figure A.6). Table 3.4 shows another example with better performance in time even spending more memory than *Shuffle*. The models variations are indicated by the number N of servers in the network. This model is composed by sparse matrices and identities with dimension $n_i = 2$ in the tensor terms. Considering that this model presents many synchronizing events, it is common the insertion of identity matrices in the tensor terms, indicating the events has no effect in the related automata. The incidence of these small identity matrices in the sparse part defined by the *cut-parameter*, does not compromises the memory a lot.

The trade-off time-memory brings a new numerical approach for a faster solution of vector-descriptor products. For the model with $N = 18$ is spent an extra memory of $\sim$17 Mb to obtain more time efficiency in the *Split* method. The computational time gains are notable when compared

Models (N)	$\mathcal{X}$	*Shuffle*			*Split*			*Sparse*		
		time (s)	size (Kb)	fpm	time (s)	size (Kb)	fpm	time (s)	size (Kb)	fpm
12	4,096	0.02110	**4.05**	368,640	**0.00330**	72.90	71,136	0.00349	900.02	**57,342**
13	8,192	0.04832	**4.70**	851,968	**0.00675**	279.06	134,240	0.00711	1,924.67	**122,878**
14	16,384	0.10892	**5.39**	1,949,696	**0.01313**	1,062.34	278,976	0.01464	4,101.36	**262,142**
15	32,768	0.24731	**6.13**	4,423,680	**0.02843**	1,287.49	579,360	0.03219	8,710.10	**557,054**
16	65,536	0.55665	**6.91**	9,961,472	**0.06395**	2,118.83	1,200,640	0.07764	18,438.88	**1,179,646**
17	131,072	1.25347	**7.75**	22,282,244	**0.14731**	6,701.31	2,514,336	0.18160	38,919.71	**2,490,366**
18	262,144	2.85996	**8.63**	49,545,224	**0.34844**	16,907.83	5,295,968	0.40969	81,928.59	**5,242,878**

Table 3.4: *First Available Server* SAN model results

to the *Shuffle* algorithm, even spending more memory unstructuring the SAN descriptor with *Split*.

Ad Hoc Wireless Sensor Network model results

This section presents the results for the *Ad Hoc Wireless Sensor Network* SAN model (Figure A.7). The numerical results were obtained for this model extending the number of automata N (nodes), then also extending the number of synchronizing events to treat as tensor product terms.

Table 3.5 shows that as the number N of nodes in the mobile chain increases, so does the computational time to solve, as well as the memory needed. Since the *Shuffle* algorithm is memory-efficient compared to the other two approaches, consequently it performs extra operations to multiply the sparse matrices in a tensor term. It is showed that the *Shuffle* method is slower than the other two methods in the models presented. These methods, on the contrary, are very time efficient despite their need for memory consuming.

Models (N)	$\mathcal{X}$	*Shuffle*			*Split*			*Sparse*		
		time (s)	size (Kb)	fpm	time (s)	size (Kb)	fpm	time (s)	size (Kb)	fpm
6	324	0.00075	**1.52**	7,344	**0.00013**	5.30	1,798	0.00025	18.93	**1,114**
8	2,916	0.00633	**2.25**	93,312	**0.00071**	103.22	14,332	0.00163	219.88	**13,928**
10	26,244	0.07454	**2.99**	1,084,752	**0.01307**	170.11	185,364	0.01520	2,510.61	**160,488**
12	236,196	0.87020	**3.72**	11,967,264	**0.18774**	1,461.79	2,230,740	0.20951	27,513.35	**1,760,616**
14	2,125,760	9.35304	**4.46**	127,545,900	**1.75147**	13,536.58	22,674,856	2.07020	292,060.08	**18,691,560**
16	19,131,900	96.44355	**5.19**	1,326,477,700	**17.99368**	118,103.25	235,251,512	21.02443	3,028,726.82	**193,838,184**

Table 3.5: *Ad Hoc Wireless Sensor Network* SAN model results

For small models ($N < 12$ nodes) the computational time and memory efficiency are reasonable enough to be dealt regardless of any algorithm in virtually any machine. It demands few more than $\sim$2 Mb in the sparse alternative and it takes less than $\sim$100 milliseconds per iteration for all algorithms. However, even in these small examples we noticed the quite impressive memory efficiency of the *Shuffle* algorithm that keeps the memory needs insignificant even for quite large models.

The remarkable result in Table 3.5 is the better time efficiency that beats even the sparse approach. Although, for each tensor product, the sparse approach could be faster for most cases, terms with many identity matrices could have a better time efficiency in *Shuffle* or *Split* algorithms. Since a SAN model is composed of many tensor product terms with *sparse* or *ultra sparse*[§] matrices, *Split* is the best option in tensor product terms where the sparse approach could be faster, but too memory demanding.

The largest model ($N = 16$) shows that *Split* is around 3 seconds faster than *Sparse*, with a memory needed of little more than 100 Mb, rather than 3 Gb needed by the sparse solution, *i.e.*, *Split* takes for this case almost 30 times less memory and still improves the time efficiency compared to *Sparse*. It is important to observe, as well, that this model has a considerable product state space of more than 19 million states. Such large model could be nearly intractable if a time and memory efficient solution is not found.

It is also noticeable that the number of floating point multiplications computed to each algorithm is not relevant to indicate a better performance in time since observations indicate that allocations/deallocations have considerable influence on the algorithms performance.

Master-Slave Parallel Algorithm model results

This section presents the results for the *Master-Slave Parallel Algorithm* SAN model (Figure A.8). This model was extended to run experiments for different numbers of N slaves and the buffer was fixed with forty positions ($K = 40$) meaning that the tensor terms are composed by matrices with dimensions given by $n_i = 3$ (representing $i = 1 \ldots S$ slaves) and a matrix with dimension $n_K = 41$ (the buffer with an empty position).

Models (N)	$\mathcal{X}$	*Shuffle*			*Split*			*Sparse*		
		time (s)	size (Kb)	fpm	time (s)	size (Kb)	fpm	time (s)	size (Kb)	fpm
5	29,889	0.0983	**16.63**	1,797,228	**0.0223**	1,447.69	399,036	0.0254	5,229.25	**333,608**
6	89,667	0.3507	**20.31**	6,488,829	**0.0931**	3,183.29	1,742,271	0.1087	18,531.62	**1,184,724**
7	269,001	1.3178	**23.35**	22,488,916	**0.3713**	9,446.93	6,555,978	0.4039	64,222.72	**4,108,760**
8	807,003	4.5805	**26.43**	76,534,019	**1.2393**	28,236.07	23,037,480	1.3487	231,315.37	**14,802,495**
10	7,263,030	50.0752	**32.43**	847,176,190	**12.9219**	240,596.27	246,651,139	13.9086	2,363,273.71	**151,247,442**
12	65,367,200	535.2877	**38.54**	9,137,063,300	**135.9426**	2,282,495.12	2,787,370,431	147.5943	26,195,236.61	**1,676,492,676**

Table 3.6: *Master-Slave Parallel Algorithm* SAN model results

Table 3.6 shows that the *Split* algorithm once again demonstrates a better time efficiency in the results of all model extensions. In fact, it presents, in general, results a little faster than the sparse approach, *i.e.*, roughly around 10% faster in larger models.

[§]Sparse matrices are classified due to their level of sparsity as *sparse*, *ultra sparse* or *hyper sparse* in [16].

However, the memory savings obtained in this second set of examples seem less impressive than those obtained for the *Ad Hoc Wireless Sensor Network* model extensions. The *Split* method still gives a considerable reduction for the huge last example ($S = 12$) bringing the memory needs from nearly intractable 26 Gb in sparse approach to large, but tractable 2.2 Gb. Once again, it is important to keep in mind that we are dealing with a model with 65 million states of product state space, and then some significant amount of memory and computational time are expected to achieve a stationary (or transient) solution.

3.3 Conclusions and Perspectives

The main contribution of this work is the proposition of a flexible hybrid vector-descriptor algorithm. The application of the *Split* algorithm on SAN models of real problems [24] showed a good tradeoff between memory and time efficiency when compared to the traditional *Sparse* and *Shuffle* approaches. Considering that we need many iterations to calculate the final probability vector π, the memory and time spent surely can be evaluated and balanced according to the available time and computational resources. Nevertheless, it is also shown that the *Split* algorithm is flexible enough to deliver in extreme cases at least the same time efficiency as the *Sparse* approach, or, alternatively, the same memory efficiency as the *Shuffle* approach. For all experiments the *cut-parameter* σ of each tensor term is chosen by running in the first iterations, some simulations of different σ, then collecting the best times obtained in a given confidence interval, considering also the memory needed.

In the Section 3.2.2 is presented the Algorithm 3.4, where the choice of the division point in each tensor term (choice of the cut-parameter σ) can be made before starting the iterative method, running some sampling iterations for each term. This procedure can have no relevant computational cost considering the gains we can achieve after running many iterations until the convergence for the solution. The sampling can be set up to reject runs clearly not feasible such as σ entirely sparse due to memory constraints, or entirely structured if many identity matrices in the term. The identities placed for example in the sparse part only generate more AUNFS to store, while in the structured part they are certainly skipped.

Note that the research for an heuristic to automatically choose the *cut-parameter* and a well-suited permutation of matrices, for each tensor product, is a considerable research challenge. This is not a trivial task, due to the tensor product term formation and intrinsic matrices details such as dimension, total number of nonzero elements and computational cost in terms of multiplications. These parameters opens the possibility of a thorough analysis of the related theoretical computational cost. Since tensor terms can be differently formed due to the structured models we deal with, the performance can also be very dependent on the choice of matrices placed in each group. The tensor

product terms that do not have too many identity matrices, or no identities at all, can be multiplied in a sparse fashion.

However the *Shuffle* algorithm deals better with terms containing many identities because it simply jumps the execution for the next normal factor to multiply. SAN are structured models composed of tensor product terms with a reasonable number of identity matrices, *i.e.* are the most commonly encountered ones, tending to push the *Split* algorithm to the structured solution, but if the memory available is not a problem, it is better to treat them in a sparse way as much as possible. Note that models with multiple synchronizations among automata tend to obtain proportionally multiple tensor terms to treat, precisely two per event. The occurrence of these events will determine the matrices sparsity and the number of identities to be splitted.

The Table 3.7 exemplifies the computational resources spent and the gains obtained after finding well-fitted *cut-parameters* for each tensor term in the descriptors. All models presented converge with different number of iterations. The previous section showed the computational time gains and memory consumption for one single iteration. However, when dealing with practical complex models to evaluate in real-life projects they demand at first the reduction of the computational time.

Models	Total iter.	*Shuffle*		*Split*		*Sparse*	
		time	size	time	size	time	size
Dining philosophers (10)	650	1.66 min.	2.81 Kb	0.62 min.	3.94 Mb	0.73 min.	12.02 Mb
Ad hoc WSN (14)	78,029	8.45 days	4.46 Kb	1.58 days	13.22 Mb	1.87 days	285.21 Mb
Master-Slave (12)	2,568	15.91 days	38.54 Kb	4.04 days	2.18 Gb	4.39 days	24.98 Gb

Table 3.7: Iterative numerical solution gains

As we can see in Table 3.7 the increasing of memory to solve models often represents a gain in computational time, *i.e.*, not in seconds but sometimes in days of processing. The last two examples (the *Ad hoc Wireless Sensor Network* model with 14 nodes and the *Master-Slave Parallel Algorithm* model with 12 slaves) are distinct SAN models presented in the Sections 3.2.2 and 3.2.2 respectively. Both represent real applications modeled through SAN descriptions and show the expressive gains obtained in terms of processing time using the *Split* approach.

Additionally, it is also possible to foresee an even more complex analysis that considers not only a sequential version of the *Split* algorithm, but also parallel implementations. For the sequential version, memory and time efficiency are dealt as a single demand, but parallel implementations should consider the amount of memory needed, volume of data exchanged and processing demands to be as evenly as possible distributed among parallel machines. Obviously, this further analysis is much more deep and complex since neither the number of floating point multiplications, nor any other known index for that matter, seems to be a good estimation of processing time.

In future researches the *Split* algorithm could be enhanced with considerations about the impact of functional elements (with their particular dependencies) in the descriptor, since it is a new advance starting to emerge also for other formalisms [45]. A similar work about these functional dependencies changed completely the performance of the *Shuffle* algorithm [37] when the tensor terms take advantage of generalized tensor algebra [37]. It is only natural to estimate that similar computational gains with functional dependencies analysis and possible automata permutations could benefit from the *Split* algorithm as well. At least the results for descriptors constructed with classical tensor algebra allow us to notice that the *Split* algorithm is already a better choice for practical vector-descriptor products.

Despite of that, the bottleneck imposed for vector-descriptor product in terms of memory is always the storage of the probability vector π (including the storage of the auxiliary vectors υ) of the same size of the product state space, except on optimized implementations using sparse vectors [7]. The *Split* method does not focus on this problem, it was proposed to speedup the iterative method based on the vector-descriptor product operation.

The probability vector π obtained using iterative methods is the basis from models measurements such as steady-state probabilities of global states or performance indexes calculations. Using vector-descriptor products the only way to obtain specific measures about a model is generating the huge vectors until they reach the memory bound imposed by the current technologies.

Focusing on state space explosion, which implies to deal with an equation system very large to be solved in a timely manner, the traditional tensorial SAN solution can be replaced by another kind of structural exploitation for example a simulation-based approach.

Chapter 4

Event-based Descriptor Solution

Real life complex systems normally are composed of many components with massive state spaces. As mentioned before, SPN [1], SAN [55] and PEPA [44, 45] were proposed to cope with the problem of the state space explosion and consequently to handle properly the matrix storage problems. However, this problem is still very challenging and it is not yet possible to analytically solve such huge models, even with advanced numerical methods such as those presented on Chapter 3. An alternative to enable the numerical solution of large models is the use of simulation techniques.

Simulation is a widely used and increasingly popular method for studying complex systems [48]. Discrete-event simulation approaches [10, 46, 47, 57] are often used to estimate an approximation of the steady-state behavior of systems, providing samples of the stationary distribution π for later statistical analysis. Note that new trends for numerical solutions were proposed in recent years, including alternatives combining numerical methods and simulation approaches [15, 14] for Markov chains. These techniques were also adapted to structured representations, but still remains the problem of having a large state space to store, and a considerable complexity to manipulate multiple components.

In the context of the SAN formalism, systems are described by somewhat independent components called automata, and each one can have interdependencies given by synchronizing or functional transitions (refer to Chapter 2). Since in discrete-event simulation [42] the system dynamics is represented only by its events and transitions effects, we could also extend this kind of description to structured models (Section 2.3.2). First approaches to simulate SAN focused on the network dynamics, adapting the model characteristics as a simulation kernel, for example, road traffic simulation [57]. Such model-driven approach was implemented as a hierarchy of uniformized events based on the automata description, starting from a predefined global state, running forward steps.

This chapter introduces forward simulation (Section 4.1) discussing the advantages and disadvantages of this approach in general. This background is needed to understand the contribution of

applying advanced simulation techniques (Section 4.2 and 4.3), such as the *Coupling from the Past* [56] to an approximated numerical solution for SAN, where the state space is intractable.

4.1 Forward Simulation

The classical event-driven simulation technique, *Markov chain Monte Carlo* (MCMC) or simply *Forward* simulation, considers the dynamic of the system defined by an arbitrary initial state and the application of a sequence of events generating a *trajectory*. The simulation estimates the system steady-state on a long run trajectory via the ergodic theorem [43, 58], which states that the system reached its stationary regime after a fixed amount of steps, *i.e.*, the estimation of the stationary distribution of being in a state $\tilde{s}$ is given by the proportion of passages through this state.

The time τ^* consists of what is known as the *warm-up* period, *transient* period or *burn-in time*. The *warm-up* period indicates the time when is still favorable to delay the sample collection, because the transient period produces unreliable samples. This means that the effects of the initial conditions have yet significance in the trajectory evolution. When it becomes insignificant, the simulation has reached the stationary regime. However, the drawbacks of this approach are mainly the *warm-up* period which is empirically fixed from an arbitrary initial state, and the fact that will certainly generate bias samples [43, 48, 58].

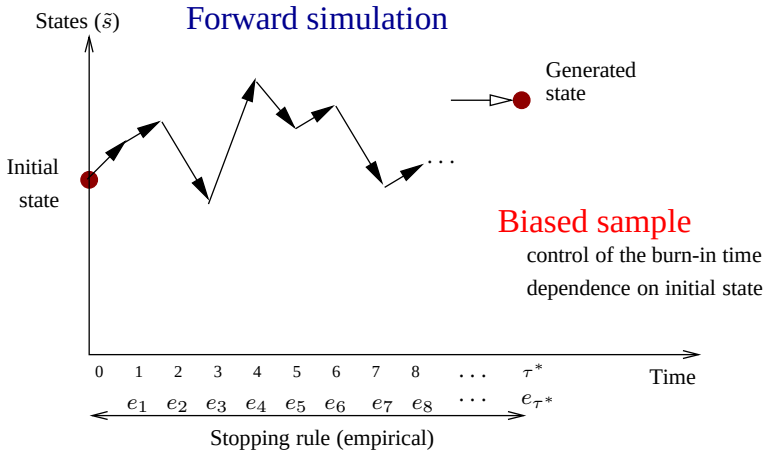


Figure 4.1: Illustration of a forward trajectory

Section 2.3.2 already described the system evolution as transition functions defined by $\Phi(\tilde{s}, e_i) = \tilde{r}$, since $\tilde{s}$ is a state in X^R, and after the firing of an event e_i, one new state $\tilde{r}$ in X^R is achieved. A trajectory is traced running forward steps through the application of uniformized events, which are randomly chosen and successively applied to a state (*e.g.*, following the distribution given by a tran-

sition matrix) then obtaining, in each application, a new state $\tilde{s}$ (when an event could not be applied to one state $\tilde{s}$ the state is not changed doing a skip operation). The last state in a trajectory of length τ^* is considered a *sample* from the stationary distribution π, although it can be potentially biased. Figure 4.1 exemplifies a trajectory generated when simulating the events application in forward steps.

An initial state $\tilde{s}_0$ is established (Algorithm 4.1, line 2) and from it, the events are applied changing the current state $\tilde{s}$ until reached the number of steps defined as length by the trajectory (line 7). The sample is then collected (line 8).

Algorithm 4.1 Forward simulation

```
1: repeat
2:    s̃ ← s̃_0 { choice of the initial state inside X^R}
3:    repeat
4:       e ← Generate-event( )
5:       { random generation of e according the distribution (α_1/α ... α_P/α)}
6:       s̃ ← Φ(s̃, e) { computation of next state s̃_{n+1} according to event e}
7:    until stopping criteria { pre-defined trajectory steps }
8:    return s̃ { generated sample is a state s̃ }
9: until stopping criteria {pre-defined number ν of samples, π calculation}
```

The problem of the establishment of how many samples are needed to obtain a confident distribution π of states probabilities is still under research (Algorithm 4.1, line 9 indicates ν as the number of desired samples to collect and calculate π). The steady-state condition in simulations may be tested using statistical confidence intervals [58].

The complexity C_s (Equation 4.1) to generate a sample in this approach is dependent of the $\Phi(\tilde{s}, e)$ (transition function) complexity cost c_Φ and the simulation time τ^* (*e.g.*, a pre-defined trajectory size). The transition function complexity is dependent of the model characteristics such as the size of automata, the number of events in each transition, and consequently, the chosen implementation.

$$C_s = c_\Phi \times \tau^* \tag{4.1}$$

Next section introduces a different advanced technique to overcome all main drawbacks of forward simulations called *Backward Coupling* simulation and follows with its application in the SAN context pointing out the characteristics of the simulation core.

4.2 Backward Coupling Simulation

A new algorithmic solution based on the *Coupling from the Past** (CFTP) algorithm, proposed by Propp and Wilson [56], overcomes the *burn-in* time problem and guarantees unbiased samples. CFTP or simply *exact* simulation, can generate a sample from the stationary distribution based on the concept of *coupling* or *coalescence* of trajectories.

Coupling has been used in many ways in Markov chains analysis [51]. In the context of this work, coupling is related to trajectories having the same sequence of events applied until their arrival in a common state. The *coupling time* τ occurs when trajectories coalesced in a given state $\tilde{s}$. Then the general principle of CFTP is the execution of trajectories in parallel starting from all states of the model. At each simulation step an event is applied to all current states. In a given time $-t$ (or τ), the transitions lead the system to the same state, and this state is a sample which can be collected. The establishment of an initial state and its dependence during the *burn-in time* in traditional forward simulations are not a problem anymore using backward couplings.

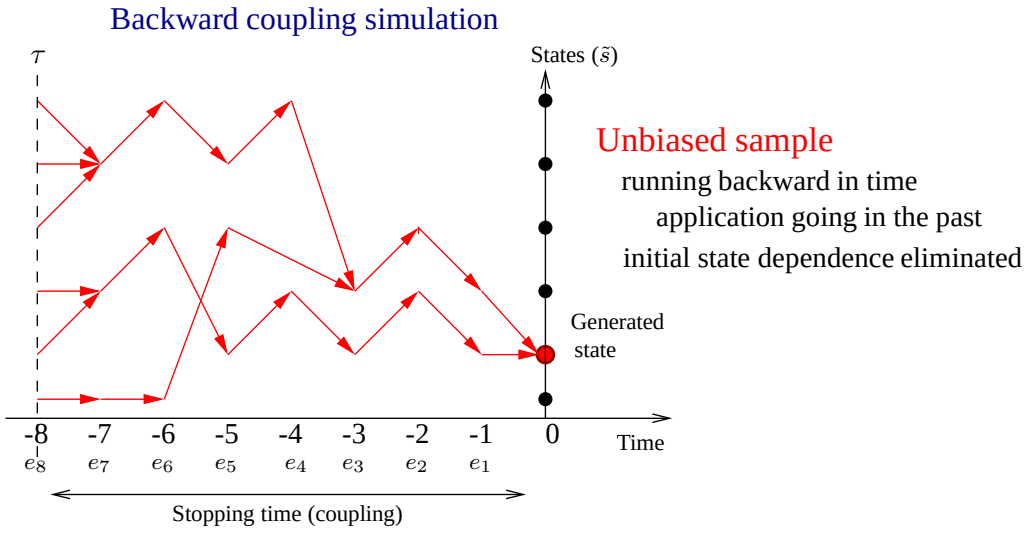


Figure 4.2: Illustration of a backward coupling of trajectories

In the Figure 4.2 all trajectories issued from all states at time -8 coupled in a state at time 0. Since the coupling time τ of the backward scheme is almost surely finite [51, 56], the scheme provides a sample distributed according the steady-state because, in other words, the method determines automatically when to stop and collect samples.

The average complexity C_s (Equation 4.2) to generate a sample in this approach is dependent of

*This simulation technique is known as *perfect* sampling or also *exact* sampling because enables us to compute samples exactly distributed according to the stationary distribution of the Markov process, *i.e.*, produces unbiased samples.

the number of trajectories in parallel, *i.e.*, the cardinality of $\mathcal{X}^R$, the average coupling time $\mathbb{E}\tau$ and the $\Phi(\tilde{s}, e)$ (transition function) complexity cost c_Φ. The transition function complexity c_Φ is the same pointed out in the forward simulation section, considering now realizations (firings) in backward steps.

$$C_s = |\mathcal{X}^R| \times \mathbb{E}\tau \times c_\Phi \tag{4.2}$$

4.2.1 SAN perfect sampling

The SAN formalism presents an underlying Markov chain, so the application of perfect sampling techniques takes advantage of the network dynamics to solve models and obtain the probabilities of global states in the stationary distribution. The backward coupling simulation states that a SAN model must be *well-formed*, consequently the model must produce only non absorbent global states as initial states for the simulation algorithm. The number of trajectories running in parallel can be at maximum equal to the cardinality of the $\mathcal{X}^R$ set, and the backward scheme will evolve, going to the past, until their coalescence as explained in Section 4.2.

Algorithm 4.2 SAN backward coupling simulation

1: **for all** $\tilde{s} \in \mathcal{X}^R$ **do**
2: $\omega(\tilde{s}) \leftarrow \tilde{s}$ {initializing trajectories with global states}
3: **end for**
4: **repeat**
5: e $\leftarrow$ Generate-event( ) {random generation of e according the distribution $(\frac{\alpha_1}{\alpha} \ldots \frac{\alpha_\mathcal{E}}{\alpha})$}
6: $\tilde{\omega} \leftarrow \omega$ { saving the states of each trajectory}
7: {computing $\omega(\tilde{s})$ at time 0 of trajectory issued from the global state $\tilde{s}$ at time τ}
8: **for all** $\tilde{s} \in \mathcal{X}^R$ **do**
9: $\omega(\tilde{s}) \leftarrow \tilde{\omega}(\Phi(\tilde{s}, e))$
10: **end for**
11: **until** global states are equal in all trajectories
12: return $\omega(\tilde{s})$ {generated sample is a global state}

Given a set of states $\mathcal{X}^R$, a set $\mathcal{E}$ of events, and the transition function $\Phi : \mathcal{X}^R \times \mathcal{E} \rightarrow \mathcal{X}^R$, *backward coupling* occurs when issuing from all states in $\mathcal{X}^R$, the trajectories couple in a state $\tilde{s}$ for a given sequence of events $\{e_n\}_{n\in\mathbb{N}}$ going to the past in time. The sample (state) collected is surely drawn from the stationary distribution. Algorithm 4.2 based on the Propp and Wilson technique [56] can be used to solve any structured representations of Markov chains, such as the SAN formalism. It demands the list of states in $\mathcal{X}^R$ as initial states for the parallel trajectories. Structurally, the trajectories are represented as a vector ω which is initialized with the global states $\tilde{s} \in \mathcal{X}^R$, supposing

a well-formed SAN model (line 2). A vector $\tilde{\omega}$ stores a copy of the current states of trajectories at each iteration.

An event e is generated (Algorithm 4.2, line 5) and the related transition function $\Phi(\tilde{s}, e)$ is applied to the current global states placed in ω positions (line $8-10$). Note that a global state is in fact a vector of size K containing the automata local states $s^{(k)}$. The event e is applied to each local state through the function $\phi(s^{(k)}, e)$ updating it (refer to Section 2.3.2). At the end, the new global state is updated too, indexing the vector $\tilde{\omega}$ which contains the earlier ω stored. This process is called backward coupling because we compute $\omega(\tilde{s})$ at time 0 of trajectory issued from $\tilde{s}$ at time $-t$. The routine will be repeated until all positions of vector ω have the same resulting state $\tilde{s}$, *i.e.*, all trajectories running in parallel have coupled (line 11). The sample of each iteration is then collected for later statistical analysis (line 12).

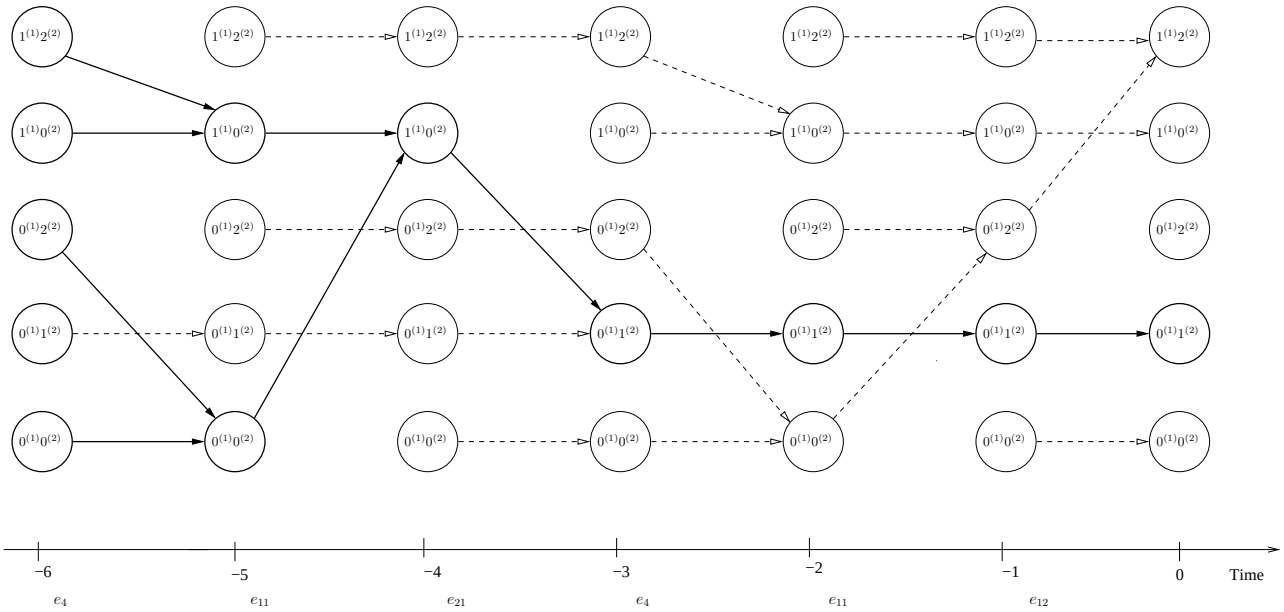


Figure 4.3: Backward coupling in 6 iterations for the SAN example in Figure 2.2

Figure 4.3 exemplifies a backward coupling which generates an exact sample in few steps, *i.e.*, it outputs a global state for the SAN example in the Figure 2.2 (Section 2.2). The process begins firing an event each time (starting in time 0 and then backwards), for all states in $\mathcal{X}^R$. The application results (the new states achieved) following the transition function definition on Table 2.2 (Section 2.3.2) are plotted in backward steps (from the right to the left). The example shows that all trajectories issued from the global states ($\{0^{(1)}; 0^{(2)}\}$, $\{0^{(1)}; 1^{(2)}\}$, $\{0^{(1)}; 2^{(2)}\}$, $\{1^{(1)}; 0^{(2)}\}$, $\{1^{(1)}; 2^{(2)}\}$) at time -6 coupled in the state $\{0^{(1)}; 1^{(2)}\}$ at time 0.

The memory needed for running the algorithm considers the storage of vectors of $|\mathcal{X}^R|$ states, *i.e.*,

one for coupling trajectories and another for collecting samples statistics. The needed size in memory is given by $2|\mathcal{X}^R|$. Simulation methods can bring a memory optimization to solve models when compared to traditional iterative numerical solutions. Iterative solvers spent at least $3|\mathcal{X}|$ (two vectors for the iterative method and one to store the stationary probabilities) except considering sparse implementations where the vectors size can be very reduced [7], mainly for models when the reachable state space is considerably smaller than $\mathcal{X}$.

Searching for new solutions for models whose are not possible to obtain a numerical solution due to the size of product state space $\mathcal{X}$, we combined structured Markovian models such as SAN, with perfect sampling techniques. However, even naturally contracting the state space using only the reachable state space in the proposed simulation solution, the size of $\mathcal{X}^R$ can also become a bottleneck for backward couplings. This lead us to exploit the product state space and monotonicity properties to reduce the number of trajectories in parallel and obtain more flexibility solving huge models.

4.3 Monotone Backward Coupling Simulation

The size of $\mathcal{X}^R$ can be exponential according to the SAN model and it can be difficult to generate and really huge to deal with. It becomes a limitation for backward coupling methods in terms of coupling time for some models, basically because it is needed one simulation per state in the model. This section introduces concepts of monotone backward coupling and partial ordering of states. Recent studies showed that the monotonicity property is fundamental for improving the efficiency of backward couplings to solve Markovian systems [10, 66, 64] since it allows the reduction of the number of trajectories in parallel, through the establishment and discovery of extremal states.

Propp and Wilson [56] have shown that for monotone Markovian models with an ordered state space it is needed to run only two trajectories in parallel: one starting from the largest state and other for the smallest state. So the algorithm operate most efficiently when the state space is a *lattice*[†] and a monotonicity condition for the events holds.

Definition. An event $e_p \in \xi$ is said to be monotone if it preserves the partial ordering ($\prec$ order) on $\mathcal{X}$. That is $\forall(\tilde{s}, \tilde{s}') \in \mathcal{X} \quad \tilde{s} \prec \tilde{s}' \implies \Phi(\tilde{s}, e_p) \prec \Phi(\tilde{s}', e_p)$. If all events are monotone, the global system is said to be monotone.

Definition. Given two global states $\tilde{s}_1, \tilde{s}_2 \in \mathcal{X}$, a state $\tilde{s}_1$ is minimal if there exists a state $\tilde{s}_2$ such that $\tilde{s}_2 \leq \tilde{s}_1$ then $\tilde{s}_2 = \tilde{s}_1$. Then $\tilde{s}_1 \in \mathcal{X}^{\min}$. Analogously, given states $\tilde{s}_3, \tilde{s}_4 \in \mathcal{X}$, a state $\tilde{s}_3$ is maximal if there exists a state $\tilde{s}_4$ such that $\tilde{s}_4 \geq \tilde{s}_3$ then $\tilde{s}_4 = \tilde{s}_3$. Then $\tilde{s}_3 \in \mathcal{X}^{\max}$.

[†]A lattice is a partially ordered set (also called a poset) in which every pair of elements has a unique supremum (the least upper bound of elements) and an infimum (the greatest lower bound) [25].

Definition. The extremal set $\mathcal{X}^M$ is a set composed by maximal and minimal states in a partially ordered $\mathcal{X}$. Then $\mathcal{X}^M = \mathcal{X}^{\max} \cup \mathcal{X}^{\min}$.

Suppose a sequence of events $e = \{e_n\}_{n\in\mathbb{N}}$, given any partial order of $\mathcal{X}$, and consequently an extremal set $\mathcal{X}^M$, if all trajectories issued from $\mathcal{X}^M$ coupled at time 0, then they will also coalesce for all states [56] in $\mathcal{X}^R$.

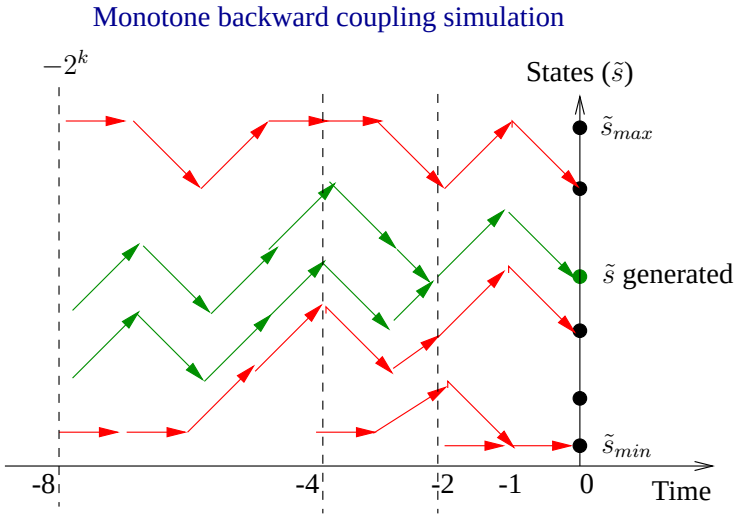


Figure 4.4: Illustration of a monotone backward coupling of trajectories

Figure 4.4 illustrates the general behavior of a monotone backward coupling, where the extremal set $\mathcal{X}^M \subseteq \mathcal{X}^R$ has two states (one maximal and other minimal), $|\mathcal{X}^M| = 2$. Going back to the past, step by step, until trajectories coalesce at time 0, all trajectories issued from $\tilde{s}_{max}$ and $\tilde{s}_{min}$ are computed in k steps from time -2^k to 0.

The simplest form of the monotone backward coupling method considers only two extremal states $\tilde{s}_{max}$ (*upper* state) and $\tilde{s}_{min}$ (*lower* state) to start trajectories at time $-t$. If the trajectories did not coalesce by time 0, a new value for t is chosen, then restarting the simulation from the new time $-t$. The coupling scheme will preserve the ordering just reusing the same sequence of events already generated, generating more events to complete the current iteration.

The simulation continues until the coupling of both chains at time 0. The application of this algorithm tries to estimate successively the value t (number of backward steps) for coupling as $t = 1, 2, 4, 8, \ldots$ until find a $t = 2^k$ when coupling occurs on simulation time 0. This adaptative step size is called *doubling scheme*, *i.e.*, at each step in the past, the length of the step is multiplied by 2 (Algorithm 4.3).

Algorithm 4.3 General monotone backward coupling with a doubling scheme

1: $t \leftarrow 1$
2: **repeat**
3: upper $\leftarrow \tilde{s}_{max}$
4: lower $\leftarrow \tilde{s}_{min}$
5: **for** $i = -t$ **to** -1 **do**
6: upper $\leftarrow \Phi(\text{upper}, e_i)$
7: lower $\leftarrow \Phi(\text{lower}, e_i)$
8: **end for**
9: $t \leftarrow 2t$
10: **until** upper = lower

The average complexity to generate a sample in this approach (Equation 4.3) is mainly dependent of the number of trajectories started from extremal states, *i.e.*, the cardinality of $\mathcal{X}^M$. The complexity considers the average coupling time $\mathbb{E}\tau$ and the $\Phi(\tilde{s}, e)$ complexity cost c_Φ.

$$C_s = |\mathcal{X}^M| \times 2\mathbb{E}\tau \times c_\Phi \tag{4.3}$$

4.3.1 SAN monotone perfect sampling

We know that the samples computation may be reduced by drawing only trajectories issued from the set of extremal states when events are monotonous. If we know $\mathcal{X}^M$ it is possible to run a monotone version of the backward coupling algorithm [38].

Adapting the Algorithm 4.3 for the SAN context we compute trajectories starting from the extremal states using a coupling vector ω of size related to the cardinality of $\mathcal{X}^M$. This algorithm have the same convergence properties as Algorithm 4.2 but can present a better coupling time for monotonous systems multiplied by a factor of $|\mathcal{X}^M|$ (which can be small enough to improve the simulation time). Also, regarding the reuse of events at each iteration (*doubling scheme*), it is needed to maintain the events generated through the whole trajectory [56]. Algorithm 4.4 uses a vector E of generated events (line 2) increased at each period (line 4). At each step in the past, the *coupling time* τ (*i.e.*, the length of the step) is multiplied by 2 (line 4).

The memory needed for running the monotone algorithm is highly related to the cardinality of $\mathcal{X}^M$ (which states the size of the coupling vector) and the size of $\mathcal{X}^R$ (size of the vector to collect samples statistics), *i.e.*, the needed size in memory is given by $|\mathcal{X}^M| + |\mathcal{X}^R|$.

Algorithm 4.4 SAN monotone backward coupling simulation

1: $n = 1$
2: $E[1] \leftarrow$ Generate-event( ) { array E stores the backward sequence of events}
3: **repeat**
4: $n \leftarrow 2n$ { doubling scheme}
5: **for each** $\tilde{s} \in \mathcal{X}^M$ **do**
6: $\omega(\tilde{s}) \leftarrow \tilde{s}$ { initial states at time $-n$}
7: **end for**
8: **for** $i = n$ downto $(\frac{n}{2} + 1)$ **do**
9: $E[i] \leftarrow$ Generate-event( ) { generate events from $(-\frac{n}{2}+1)$ to $-n$, events from -1 to $(-\frac{n}{2}+1)$ have been generated in a previous loop}
10: **end for**
11: **for** $i = n$ downto 1 **do**
12: **for each** $\tilde{s} \in \mathcal{X}^M$ **do**
13: { $\omega(\tilde{s})$ is the state at time $(-i-1)$ of the trajectories issued from $\tilde{s}$ at time $-n$}
14: $\omega(\tilde{s}) \leftarrow \Phi(\omega(\tilde{s}), E[i])$
15: **end for**
16: **end for**
17: **until** all positions of vector ω are equal
18: return $\omega(\tilde{s})$ { generated sample is the global state in $\omega(\tilde{s})$}

4.3.2 Extremal global states extraction

Glasserman and Yao [42] investigated the search for partial (and total) ordering in discrete-event models looking at their own structure, naturally retaining the order in which states in the chain are accessed firing the respective events. This procedure incrementally generates a *feasible* set (set of trajectories), until all states are accessed (total ordering), or a given partial ordering is identified. However, the search for an order regarding feasible sets could have a high enhanced computational cost for huge models because one must look at all possible trajectories starting from $\mathcal{X}^R$ to effectively begin to search extremal states.

It is a fact that the monotonicity property of events guarantees the existence of a partial ordering in which is possible to obtain the $\mathcal{X}^M$ set of extremal states [56]. So applying transition functions over each state in $\mathcal{X}^R$, retaining the new states achieved one can just verify if these new ones are greater (not yet accessed in some way) than the source state in the firing process. In this case is not necessary to retain the order of access of these new states as feasible sets do [42].

In a SAN context, the *extremal* set $\mathcal{X}^M$ is composed of global states where there is no greater state achieved in $\mathcal{X}^R$ than itself in the underlying chain, after the firing of all events in ξ. The constructive algorithm proposed (Algorithm 4.5) analyzes each reachable state of the model (line 20), firing the events of the set ξ (line 5) and storing the generated (achieved) states at each firing (line 10). The

Algorithm 4.5 Extremal set for SAN models with component-wise formation

1: $M[0] \leftarrow$ Add($\tilde{s}_{min}$); { list of accessed states in the $\mathcal{X}^R$, initially storing the state $\tilde{s}_{min}$}
2: cState $\leftarrow M[i]$; { it indicates the current observed state in the list M, initially $i = 0$}
3: **repeat**
4: isExtremal $\leftarrow$ true;
5: **for all** $e_p \in \xi$ **do**
6: { events firing over the current observed state}
7: nState $\leftarrow \Phi$(cState ,e_p);
8: **if** (nState $\notin \{M[0], \ldots, M[cState]\}$) **then**
9: { if it is not already accessed, it adds new state to the list M and it is not an extremal}
10: $M \leftarrow$ Add(nState);
11: isExtremal $\leftarrow$ false;
12: **end if**
13: **end for**
14: { if no nState are added to the list M, cState is an extremal}
15: **if** (isExtremal) **then**
16: $\mathcal{X}^M \leftarrow$ Add(cState); { it adds cState to $\mathcal{X}^M$ }
17: **end if**
18: $i = i + 1$; { it goes to the next element in M to analyze firings}
19: cState $\leftarrow M[i]$; { it updates current state}
20: **until** $i = |\mathcal{X}^R|$ { the condition is to access all reachable states in the model once}
21: return extremal set $\mathcal{X}^M$; { the extremal set is completed }

search for extremal elements starts from an initial state $\tilde{s}_{min}$ (line 1) defined by a component-wise ordering already known (which can be simply a lexicographical order).

It is important to notice that event firings from a state can lead to the same state (or reachable states already accessed). In this case, the state observed can be an extremal state (lines $14 - 17$), *i.e.*, the classification as an extremal state[‡] happens if the state does not generate any new state, meaning it is before accessed with the firing of all possible events in the model. Note that, by the same principle, transitions fired from the established minimal state do not achieve states lower than itself, considering we already started from the canonical minimum.

The complexity to find the extremal set using Algorithm 4.5 is given by the cardinality of $\mathcal{X}^R$ and is dependent of the number of events $|\xi|$ in the model (Equation 4.4).

$$|\mathcal{X}^R| \times |\xi| \tag{4.4}$$

Supposing a queueing network (QN) with two queues (Figure A.1) with capacities K_1 and K_2 respectively. This component-wise model where the behavior of both queues is equivalent, has arrivals

[‡]Markovian Free-choice Petri nets consider as extremal states the blocking markings of event graphs [10].

and departures of clients, finite queues capacities, and clients loss if queues are full. The initial global state considered is both queues empty. The transition function $\Phi(\tilde{s}, e_p)$ which describes this behavior considering global states $\tilde{s} = \{s^{(1)}; s^{(2)}\}$ configurations can be given by:

$$\begin{aligned}
\Phi(\{s^{(1)}; s^{(2)}\}, e_1) &= \{s^{(1)} + 1; s^{(2)}\} \\
\Phi(\{K_1^{(1)}; s^{(2)}\}, e_1) &= \{K_1^{(1)}; s^{(2)}\} \\
\Phi(\{s^{(1)}; s^{(2)}\}, e_{12}) &= \{s^{(1)} - 1; s^{(2)} + 1\} \\
\Phi(\{s^{(0)}; K_2^{(2)}\}, e_{12}) &= \{s^{(0)}; K_2^{(2)}\} \\
\Phi(\{s^{(1)}; s^{(2)}\}, e_2) &= \{s^{(1)}; s^{(2)} - 1\}
\end{aligned}$$

Starting from $\{0^{(1)}; 0^{(2)}\}$, or simply $\{00\}$, one can construct the extremal set $\mathcal{X}^M$ analyzing each reachable state achieved through the firing of events over the states in $\mathcal{X}^R$. At the end, the extremal set $\mathcal{X}^M$ found for this example is composed of two states: $\{00, 22\}$.

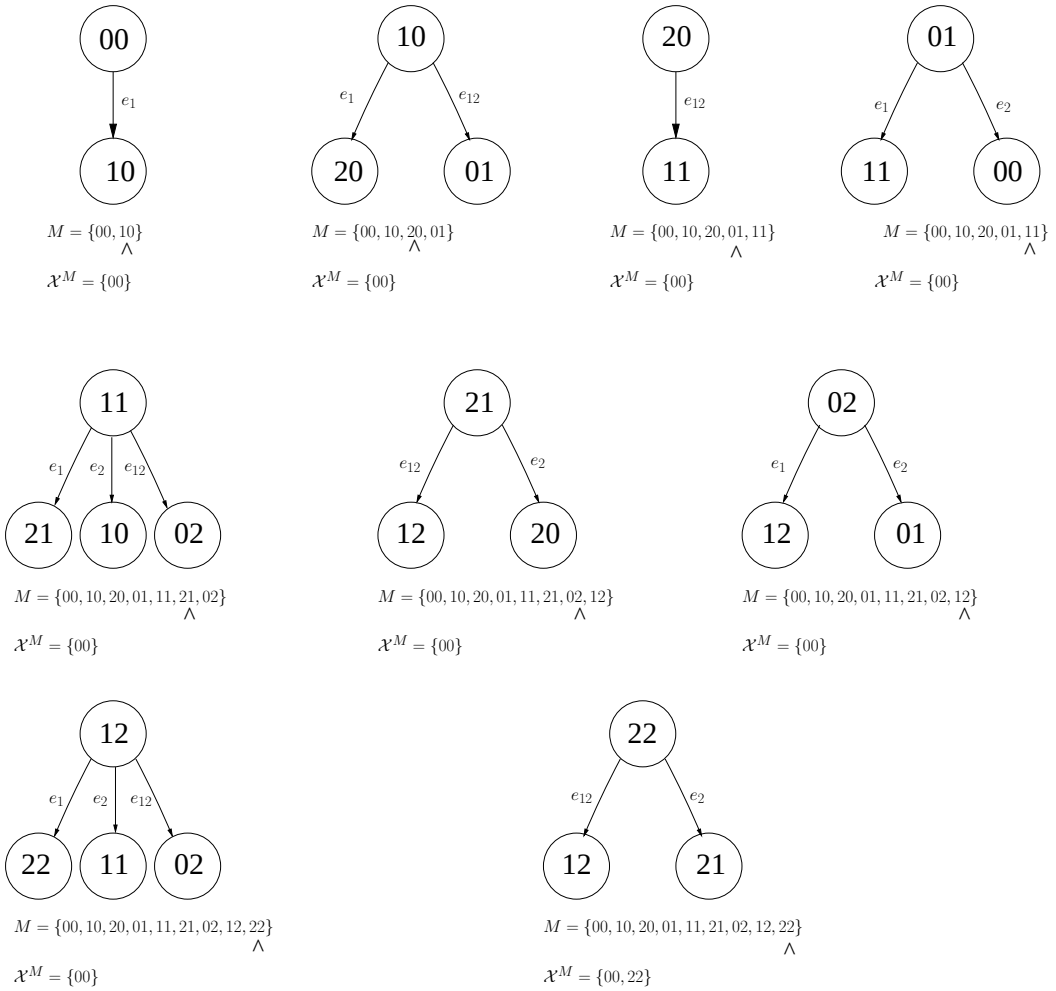


Figure 4.5: Extremal set construction for the QN model in SAN

Figure 4.5 shows step by step the formation of an extremal set (following Algorithm 4.5) for the example in Figure A.1. Next sections present a classification of SAN models, regarding the component-wise ordering, *i.e.*, the structural formation of the underlying chain. The identification of extremal states is mainly related to structural characteristics: if the chain can be viewed as a *lattice* we identify only two extremal states, else it is classified as *non-lattice* (more than two extremal global states). Note that in the absence of a component-wise ordering the models yet present the reachable set as alternative to run backward simulations and recent advances such as envelopes on the simulation of non-monotone systems could be applied [18]. In the context of this work we focus on component-wise models whose is possible to extract extremal states.

Canonical component-wise ordering in SAN

SAN models can have states naturally ordered such as the integer set, for example, the SAN models equivalent to some Markovian queueing networks [63, 64, 66] as the example presented in Section 4.3.2. When all N queues in a system are empty, the local states $s^{(i)}$ of each queue (or automaton) are equal to 0 (analogously, all queues full means automata local states equal to the queues capacities K_i, where $i = 1 \dots N$). When occurring a queue arrival (or departure) the local state is updated with $s^{(i)} + 1$ ($s^{(i)} - 1$, respectively). In this case, the partial order of the related product state space is established based on this component-wise ordering. SAN descriptions derived from monotone queueing networks can be simulated taking advantage of having only the canonical minimum (all queues empty) and maximum (all queues full) states. Then only two paths are needed to simulate a monotone backward coupling.

Definition. The *canonical component-wise ordering* means that the underlying structure of the model can be viewed as a *lattice*, *i.e.*, all global states have the same *supremum* and *infimum* states.

Given two arbitrary global states $\tilde{s}_1, \tilde{s}_2 \in \mathcal{X}$ and a defined partial order, and verifying $\tilde{s}_1 \leq \tilde{s}_2$, it is possible to define which one is the largest state [30]. The extremal states are given canonically by the first and the last state of $\mathcal{X}^R$ considering that $\mathcal{X}$ is lexicographically ordered due to the component-wise characteristic. So the events e_1, e_{12} and e_2 are monotone according canonical component-wise ordering of $\mathcal{X}$. Figure 4.6 shows successive events application until a state where the achieved state with a given event is itself (see the loops in specific states in the figure). At right, also shows the underlying chain structure as a lattice.

We can consider the minimal and maximal local states of each automaton $\mathcal{A}^{(k)}$ defined by the natural order on integer. Supposing $K_1 = 2$ and $K_2 = 3$, the maximal set can be considered $\mathcal{X}^M = \{\{0;0\},\{2;3\}\}$. The minimal local state of both automata is the state 0, and the maximal local state is 2 for automaton $A^{(1)}$, and 3 for automaton $A^{(2)}$ respectively. The application of the

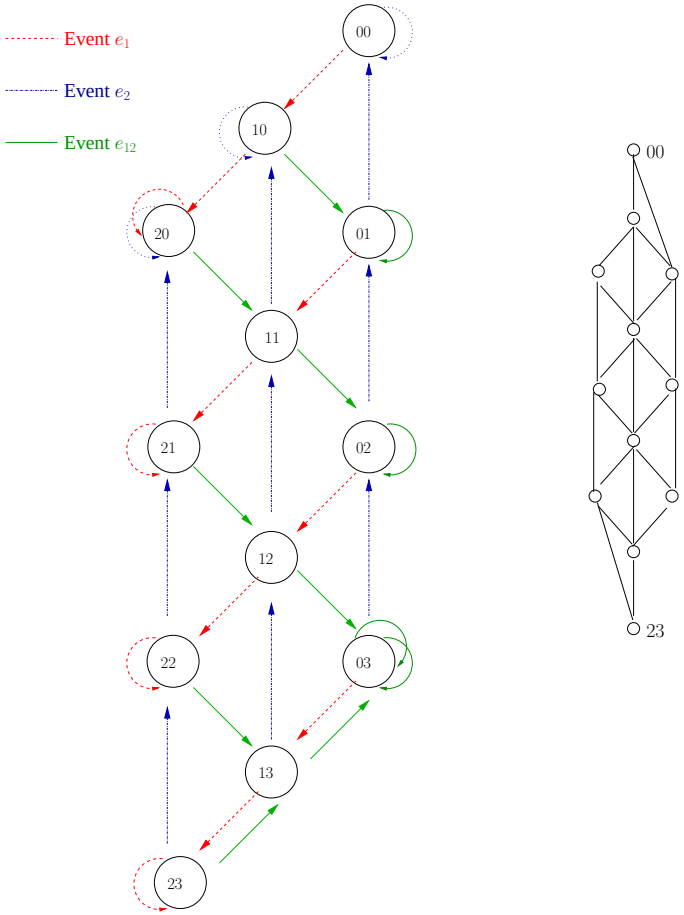


Figure 4.6: Canonical component-wise ordering for the QN model in SAN

transition function $\Phi(\tilde{s}, e_p)$, for each event $e_p \in \xi$, considering each state $\tilde{s} \in \mathcal{X}^R$ using the Algorithm 4.5, lead us to obtain the same extremal global states extracted considering the canonical ones firstly assumed.

Experimental work using SAN canonical component-wise models, showed that different state space partial orderings can be explored due to the extraction of different subsets of global states when running monotone algorithmic versions. The extremal set of global states is dependent on the partial order established. Figure 4.7 shows the Markovian structure (as seen in Figure 4.6) and a different lattice (on the right side) obtained. Now we started from the minimal local state of the first automaton combined with the maximal local state of the second automaton ($\{0; 3\}$), and achieved the maximal local state in the first automaton combined with the minimal in the second $\{2; 0\}$. More experiments must be conducted towards to different sets of extremal elements as effective as the canonical ones for monotone backward simulations.

The assumption of existing one minimum and one maximum local state per automaton which

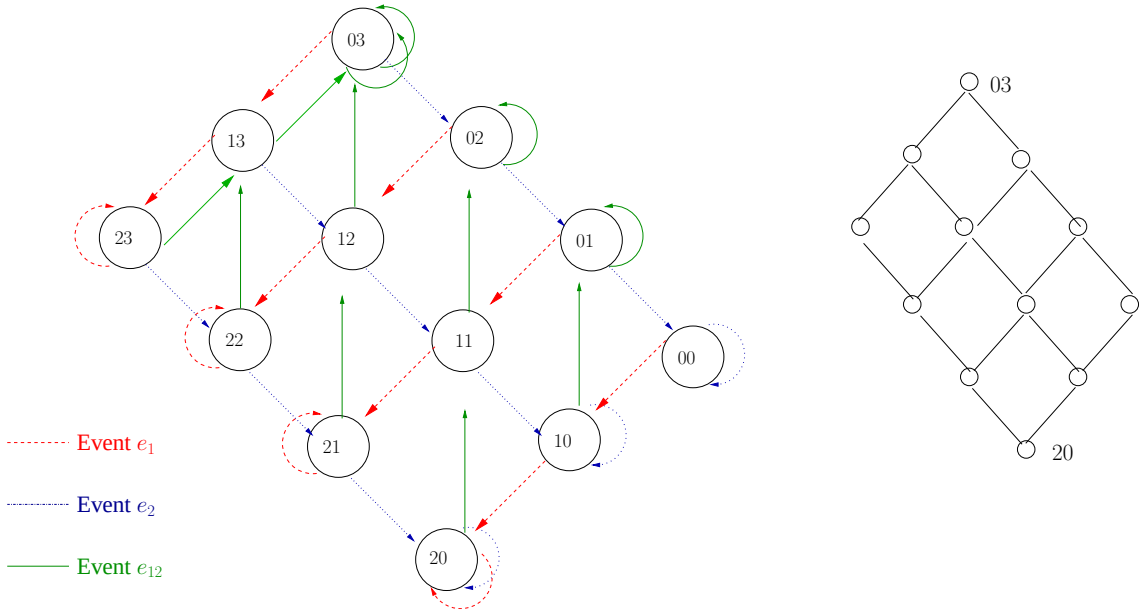


Figure 4.7: Another component-wise ordering for the QN model in SAN

guarantees the exact sampling can be applied also for huge models that follow the canonical component-wise ordering principle. For the example, the simulation could run only two trajectories in parallel: all queues empty (minimal local states $\{0;0\}$) and all queues full (maximal local states $\{K_1;K_2\}$). Other set of two extremal states could be: the first queue empty and the second full (global state $\{0;K_2\}$) and first queue full and the other one empty (global state $\{K_1;0\}$).

Non-lattice component-wise ordering in SAN

The successive firings of events for the queueing system example, when already exists a canonical formation, leads us to a *lattice* where there are only two extremal global states (Figure 4.6). However in the absence of a canonical component-wise model formation, for each event in the model, the state space partial ordering can be constructed firing events in the underlying chain structure, retaining or not the order in which the states are accessed, but mainly identifying the extremal states (Algorithm 4.5).

Considering each reachable state and all events, the last different states, in the trajectories of accessed states, are the extremal states, this means that exists a partial order for $\mathcal{X}^R \subseteq \mathcal{X}$ ($\prec$), when it is possible to compare two states for a given event $e_p \in \xi$, independent of event rates[§].

[§]There is a type of monotonicity called *stochastic* which uses the model rates [40] to establish a partial ordering of states in a Markov chain. The monotonicity we search in SAN is called *realizable* and in the literature it deals with transitions effects, *i.e.*, the states achieved on the *realizations* [39] or event firings.

Definition. The *non-lattice component-wise ordering* means that the underlying structure of the model presents a partial ordering but no canonical minimal and maximal global states, *i.e.*, a set of extremal global states is identified following successive transition function applications over $\mathcal{X}^R$.

Firstly, the component-wise ordering supposes that local states have a predefined order, then the Cartesian product of states generates automatically partially ordered global states. The states will always have transitions to greater or lower global states considering a lexicographical order. So when there is no possible transition to be fired to a greater state, this means that we have found an extremal state in the chain. The classification as *non-lattice* is used because this kind of model does not have only one infimum and supremum state, then it can not be considered a *lattice*.

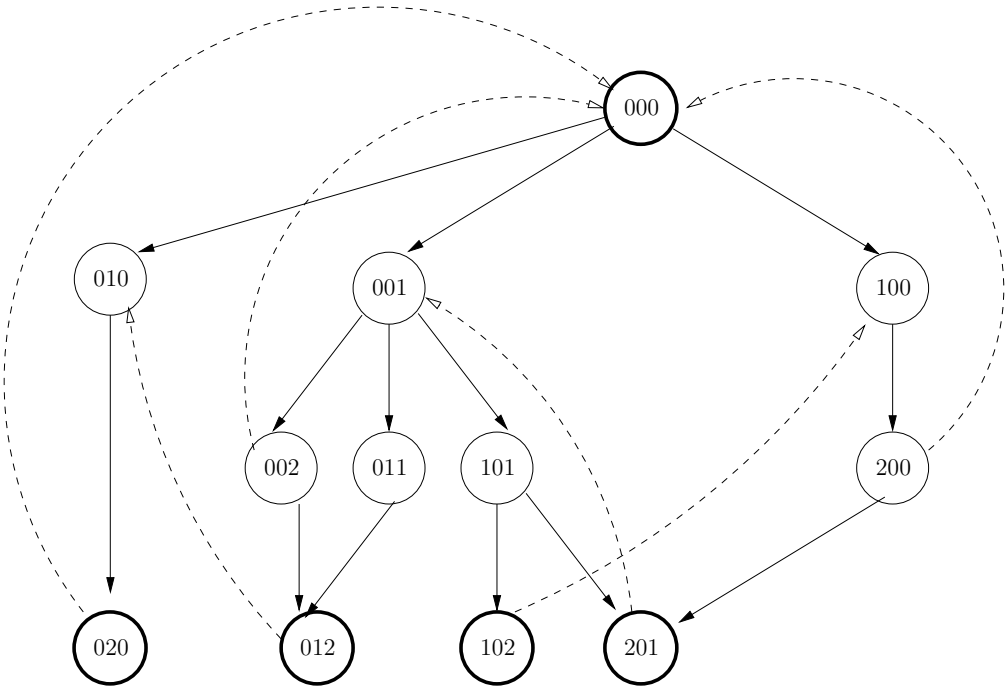


Figure 4.8: Non-lattice component-wise ordering in a model of 3 philosophers

An example of a component-wise model is the classical resource sharing with mutual exclusion presented as the *dining philosophers* on Appendix A.3. Regarding the structural formation showed in Figure 4.8 of dining philosophers with resource reservation (related to Figure A.4 with 3 philosophers), the monotonicity properties are verified for all events lt_i, tr_i, rl_i, rt_k, tl_k and lr_k since for each one remains the state space ordering in $\mathcal{X}^R$. Given the minimal global state $\{0^{(1)}; 0^{(2)}; 0^{(3)}\}$, or simply $\{000\}$ (all philosophers thinking) as initial extremal state to generate the $\mathcal{X}^M$, the other extremal states (the bold faced states $\{020\}$, $\{012\}$, $\{102\}$, $\{201\}$) are naturally the ones with greater indexes than the states they can achieve firing transitions (Algorithm 4.5). Note that the Figure 4.8 only rep-

resents the directed transitions that can be fired among states to allow the graphical visualization of the extremal states, differently of Figures 4.6 and 4.7.

Supposing now six philosophers on the dining philosophers without resource reservation (related to Figure A.5), the application of the transition function in the component-wise formation, returns the extremal states for this SAN model. Regarding structural properties of this model all events $et_k, te_k \in \xi$ are monotone since they retain the component-wise ordering of global states in the chain formed by this class of models. The $\Phi(\tilde{s}, e_p)$ application over state $\{000000\}$ (all philosophers thinking), generate states in $\mathcal{X}^R$ and so on, until no more new states are achieved using $e_p \in \xi$.

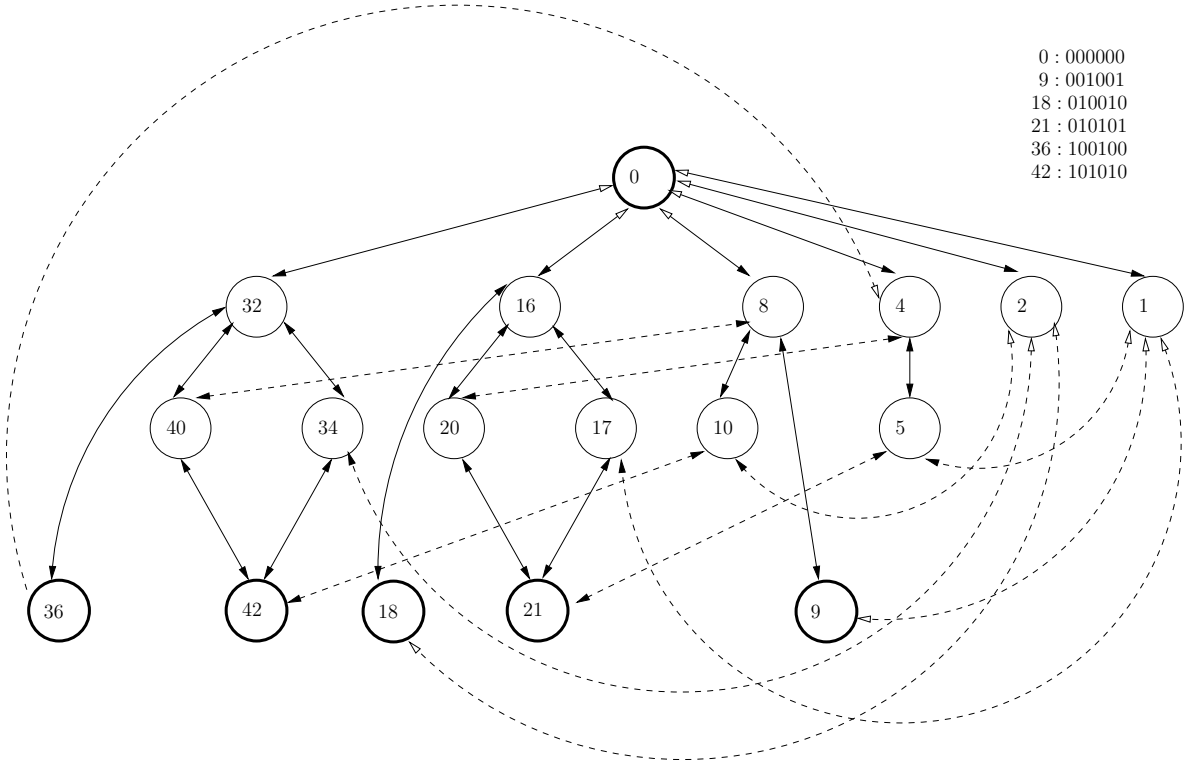


Figure 4.9: Non-lattice component-wise ordering in a model of 6 philosophers

Considering the model global states are formed simply by *bits* representing local states, so the state $T^{(k)}$ is represented by O and $E^{(k)}$ represented by 1, we have for example, a set $\mathcal{X}^R = 18$ and $\mathcal{X}^M = 6$. Figure 4.9 shows the states with their related indexes to facilitate the graphical visualization. The marked states are extremal elements for this model with six philosophers, and their respective configurations of local states are indicated at right in the figure.

This means after all, that walking in the chain, applying the transition function successively, we can reach and collect the extremal elements (Algorithm 4.5) not necessarily retaining the state space partial ordering. The canonical component-wise ordering (where $|\mathcal{X}^M| = 2$) and the non-lattice

component-wise ordering (where $|\mathcal{X}^M| < |\mathcal{X}^R|$) are SAN partial orderings well fitted to run monotone backward simulations. However it is not always easy to identify these extremal global states inside $\mathcal{X}^R$ mainly due to the reachable state space size and also the quantity of events in the model. Once they are identified, the trajectories in parallel can be drastically reduced, and consequently, the computational cost can be minimized.

Note that in the models where the classical monotonicity property is absent, the state space does not contain neither a natural minimal state nor a maximal state. Some extensions are been proposed for partilly ordered state spaces in other formalisms, *e.g.*, find extremal states for a sub-class of Petri nets [10] or for Free-choice nets [11], and simulating general non-monotone Markovian systems using lower and upper envelopes[18]. Unfortunately, there are many cases in which the monotonicity does not exist or is difficult to define. Some cases will be important to develop a method which does not require monotone structures [19]. However, perfect sampling enables us to obtain exact samples which is extremely advantageous.

4.4 Theoretical Contributions

This section presents theoretical results demonstrated through the execution of the simulation algorithms on two classical models such as the *Queueing Network* model and the *Dining Philosophers* model (Figures A.1, A.4 and A.5) mainly to validate statistically the approach. Also is shown for these examples the state space contraction through the finding of their extremal states briefly evaluating the sampling time to guide further works. The global probabilities used for statistical validation were collected directly from the probability vector obtained through the execution of the algorithms inside the *PEPS* environment.

The numerical method *Shuffle* was used to collect the stationary solution (global states probabilities) with 1.0E-10 precision which represents our expected result. Backward and monotone backward coupling implementations generate $\nu = 10^6$ samples on each run. Due to the independence of the samples we applied the central limit theorem to compute the solutions confidence intervals at level 95% considering 50 runs. The gain in state space contraction for component-wise models are showed regarding memory costs as well as a sampling time analysis varying the number of philosophers (Tables 4.1 and 4.2), showing an average time to generate an exact sample for each model with the monotone approach.

All executions have been done on an Intel Pentium dual core 3.0 GHz machine under Linux operating system with 2 Gb of memory. The modules inside the *PEPS2007* environment were compiled using `g++` compiler, with optimizations ($-O3$).

4.4.1 Statistical validation

In this section the experimental probabilities obtained by simulation are compared to the actual *PEPS* results, for models with a canonical component-wise ordering (canonical extremal states) and for the one with a non-lattice component-wise ordering (obtained extremal states). The statistical test used to compare the observed results with the expected results provided by *Split* is the *Chi-square* test[¶] (or simply Chi^2). We aim to show that there is no significant difference between the expected and the observed probabilities considering a level of significance $\alpha = 0.05$ and a degree of freedom df, for this small examples, $df = |\mathcal{X}^R| - 1$.

Queueing Network model results

Considering the example (Figure A.1) the queues rates as $\lambda_1 = 0.8$ $\lambda_2 = 0.9$ $\lambda_3 = 1.2$ the model stationary distribution provides our expected values (which is the actual *PEPS solution*) and the statistical analysis of $\nu = 10^6$ samples generated running both simulation algorithms as our observed values (backward coupling simulation and monotone backward coupling simulation).

According to the Chi-square statistic obtained for the backward coupling simulation output (Chi$^2 = 11.33$), with $\alpha = 0.05$ and degree of freedom ($df = 11$), the value 19.675 was our parameter. We find the calculated value 11.33, lieing between 5.578 and 17.275. The corresponding probability was $0.90 < P < 0.1$. This is below the conventionally accepted significance level of 0.05 or 5%, so the hypothesis that the two distributions are the same is verified.

The Chi-square statistic obtained for the monotone backward coupling simulation output (Chi$^2 = 17.31$) for same α and df lies between 17.275 and 19.675, then the corresponding probability is $0.10 < P < 0.05$. This is also below the accepted significance level, so we consider both distributions the same. We vary the capacity of both queues to perform the Chi^2 tests.

Dining Philosophers model results

Considering the example of the dining of three philosophers with resource reservation (Figure A.4) the acquisition rates are defined as 0.4 (events tr_i, rl_i, tl_K, lr_K) and the release rates are defined as 0.3 (events lt_i, rt_K). According to the Chi-square statistic obtained for the backward case (Chi$^2 = 13.18$) and the monotone backward case (Chi$^2 = 12.19$) both lied between 5.578 and 17.275. The corresponding probability is $0.90 < P < 0.1$. This is below the accepted significance level of 0.05 or 5%, so for this example we also consider both distributions the same.

[¶]Chi-square is a statistical test commonly used to compare observed data with the data we would expect to obtain, according to a specific hypothesis.

Varying the examples size, consequently analyzing other degrees of freedom for the obtained results, the verification still indicates that the distribution is the same of the expected. The next analysis to be done is related to the monotone sampling process which can be memory-efficient when compared to backward coupling approaches allowing solution of huge models.

4.4.2 SAN monotone perfect sampling analysis

Table 4.1 shows in its last lines huge models impossible to solve with the current *PEPS* software tool mainly because the size of $\mathcal{X}$, and the $\mathcal{X}$ contraction in $\mathcal{X}^M$ to run the alternative solution which is based on the perfect sampling. The costs in memory are drastically reduced since for monotone versions is stored just the coupling vector with extremal elements instead of the reachable state space. For all simulation approaches that could be used, we still need to store the frequency of coupled states, however with iterative solutions we have stored the product state space (current implementation) which already indicates a gain in the memory constraints.

K	$\mathcal{X}$	$\mathcal{X}^R$	$\mathcal{X}^M$	*PEPS* (s)	*Simulation* (s)
5	243	70	11	0.0001	0.0046±0.0009
6	729	169	17	0.0005	0.0194±0.0019
7	2187	408	27	0.0017	0.0377±0.0028
8	6,561	985	43	0.0032	0.0734±0.0051
10	59,049	5,741	111	0.0381	0.3369±0.0240
12	531,441	33,461	289	0.5513	1.5873±0.0794
14	4,782,969	195,025	755	5.7122	6.8181±0.2795
16	43,046,721	1,136,689	1,975	68.7043	27.7824±1.0808
18	387,420,489	6,625,109	5,169	n/a	108.1243±5.2696

Table 4.1: *Dining Philosophers* model (with resource reservation) - sampling results

The same remarks are consistent to the times presented here, specially due to the fact that Table 4.1 presents times for one iteration in the *PEPS* numerical solution, and one sample generation for the monotone perfect sampling. For the last model ($K = 18$) the *PEPS* solution could not be achieved since it represents a state space of more that 387 million states, which is considerably above the current overall numerical solution limitation which is 65 million states in the current implementation.

The actual number of samples needed to generate depends immensely on the numeric characteristics of the model itself. Different parameters such as the actual numeric rates of the events may change the required number of samples to achieve statistical approximation of the stationary regime. Analogously, the numbers of iterations to perform the iterative solution methods in the *PEPS* tool also

depends on such characteristics. Therefore in the Table 4.2 we indicate the amount of time needed to perform one single sample generation with the contracted state space in the simulation module, and one single iteration in the numerical solution implemented on *PEPS*.

However, the values in seconds presented here are to be considered with caution, since nothing relates the number of needed iterations in *PEPS* with the number of samples needed in our simulation tool. For example, the first model ($K = 6$) needed 528 iterations to achieve a precision of 1.0E-10 in the *PEPS* solver, while the precision achieved with 10^5 samples generated by simulation lies on approximately 1.0E-3. The examples was extended just beyond the capacity limit of *PEPS*, since the last examples ($K = 26$ to $K = 30$) are too massive to run on our target machine.

K	$\mathcal{X}$	$\mathcal{X}^R$	$\mathcal{X}^M$	*PEPS* (s)	*Simulation* (s)
10	1,024	123	18	0.0005	0.0305±0.0034
12	4,096	322	30	0.0025	0.0750±0.0059
14	16,384	843	52	0.0118	0.2109±0.0149
16	65,536	2,207	91	0.0560	0.6020±0.0435
18	262,144	5,778	159	0.2964	1.4334±0.0838
20	1,048,576	15,127	278	1.3940	3.4625±0.1822
25	33,554,432	167,761	1,131	5.1158	28.2105±1.1764
26	67,108,864	271,443	1,498	n/a	41.5225±1.9412
27	134,217,728	439,204	1,984	n/a	64.7593±2.5652
28	268,435,456	710,647	2,628	n/a	96.0582±4.2948
29	536,870,912	1,149,851	3,481	n/a	143.0277±4.8180
30	1,073,741,824	1,860,498	4,611	n/a	210.2677±7.7823

Table 4.2: *Dining Philosophers* model (without resource reservation) - sampling results

The simulation times presented in Tables 4.1 and 4.2 with their confidence intervals are estimations for a generation of one sample using perfect sampling. The range of the confidence intervals vary because different coupling times can occur in each simulation iteration. Figure 4.10 shows different coupling vector sizes and their contraction during an experiment collecting five hundred samples (using backward simulations for the philosophers with resource reservation example).

The graph represents an example of the different coupling times that can occur, and also the coupling vector reduction[‖] until a sample is collected. Note that we can have a quick memory reduction at the beginning and this specific behavior of the backward coupling occurs in many observed cases.

Future researches can exploit for example advanced transitions functions to increase the coupling probability at each backward step, reducing then the memory needs of the coupling vector which is

[‖]The current version for the simulation algorithms presents a dynamic vector for coupling control which reduces its size at each iteration. At the end, with only one position, the referred sample can be collected.

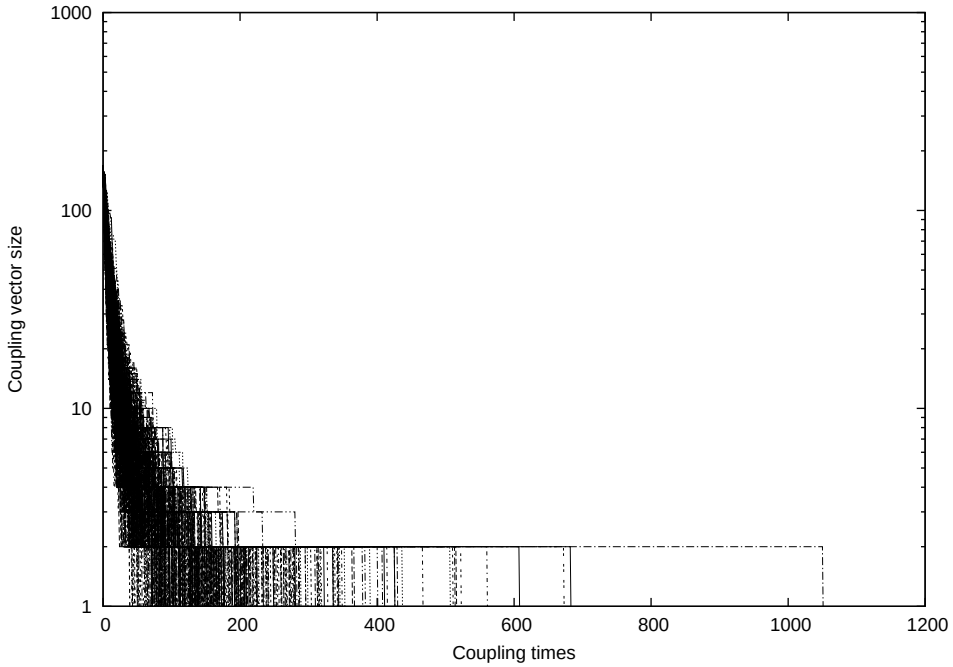


Figure 4.10: Illustration of the coupling vector reduction collecting samples

related to the number $\mathcal{X}^R$ of trajectories in parallel.

4.5 Conclusions and Perspectives

The discrete event simulation aims to reproduce the system evolution step by step studying a particular realization of a stochastic model. As discussed on Section 4.1 the advantage is that simulation is a generic approach which can be applied to every model described as a set of events and associated transition functions. The main disadvantages are related to the simulation computational cost (justifiable only if there is no analytical solution available) and the statistical validation since traditional approaches does not guarantee exact samples and it is difficult to determine values such as the burning time period, the starting state and the quantity of samples *a priori*. The backward coupling simulation provides a way to overcome the forward simulation drawbacks (the choose of an initial state, not necessary anymore since we use all states starting parallel trajectories, and the transient period determination, eliminated since there are a finite number of steps in the past to collect an exact sample). The simulation results can be presented with confidence intervals allowing the verification if the quantity of samples generated is sufficient for the model statistical analysis.

The numerical results (Section 4.4.2) pointed out that for the models in which the numerical solution is no longer possible with *Split*, perfect simulation seems to be a reasonable alternative

at least for component-wise models. The current limitation of *PEPS* software tool is in order of $\mathcal{X} \leq 6 \times 10^7$ states using 1 Gb RAM machine because it needs to store probability vectors for the state space $\mathcal{X}$. The SAN simulation approach, on the contrary, will work only with vectors of size related to the extremal set of the models and the reachable state space $\mathcal{X}^R$. Moreover the bottleneck imposed for vector-descriptor products in terms of memory needs to solve and present a solution in a timely manner (the storage of the probability vector with size equal to the product state space) is replaced by the storage of only one vector of size equal to the reachable state space and a coupling vector at least of size related to the cardinality of $\mathcal{X}^M$ in simulations. The algorithm presented does not focus on optimizing the simulation time at all, but it is a valid alternative to the lack of using iterative methods based on vector-descriptor products for huge models.

Optimizations in the transition function derived from descriptors, events uniformization techniques, coupling test optimizations are future works for the improvement of the perfect sampling on SAN. Additionally, it is also possible to foresee an even more complex algorithm analysis that considers coupling times optimization based on other model properties or different transition functions definitions. The generation of a even yet reduced extremal set $\mathcal{X}^M$, and also the notable reachable state space contraction during a backward simulation run (see Section 4.4.2), are ongoing researches for non-lattice component-wise state space formations. The algorithms could also be enhanced with recent researches in the area concerning the use of concepts of reward coupling [65] and the concept of envelopes applied for non-monotone models [18].

Due to the independence of samples we can also consider parallel simulations to overcome the time complexity related to the approach in general, therefore generating huge amounts of samples faster. Table 4.3 shows the expected gains of a parallel distribution of simulations even using non-optimized algorithms. We indicate in the first column the number of processors (*#proc*) involved to calculate 10^6 samples and the expected computational times in each case.

#proc	*execution time*
1	∼7.01 years
16	∼5.26 months
32	∼2.63 months
128	∼19.71 days

Table 4.3: Expected parallel distribution gains for simulation

The model used as example is the one indicated in the last row of the Table 4.2 which has ten philosophers (model with resources reservation) with approximately $\mathcal{X} \sim 10^9$ of state space ($\mathcal{X}^R \sim 10^5$ and $\mathcal{X}^M \sim 4 \times 10^3$). This model has no possible solution with the *Split* algorithm due to the state space explosion problem. The simulation seems to be a valid alternative mainly considering a

considerable number of processors to produce the exact samples. Note that generating more samples in more different processors is always better in this approach.

The main contribution of this chapter is the design of a perfect sampling algorithm for solving huge SAN models using backward coupling simulation [38]. Moreover we studied the monotonicity property as a key for extracting extremal global states (the set $\mathcal{X}^M$) in component-wise models where there are no canonical extremal states identified. For models with a non-lattice component-wise ordering of the underlying Markovian graph, is showed that the simulation complexity can be greatly reduced by using only $\mathcal{X}^M$ to run trajectories in parallel. A simple procedure (Algorithm 4.5) for the $\mathcal{X}^M$ extraction is presented as well as the results obtained running the monotone algorithm version (Algorithm 4.4).

The probability vector π obtained performing the statistical analysis of samples is the basis from models measurements such as steady-state probabilities of some global states or performance indexes calculations. Using simulation approaches to obtain specific measures could not demand much memory such as the vector-descriptor product where the vector π state space sized is always needed.

Chapter 5

Conclusion

This thesis focused on the major challenge of the modeling formalisms which is the impact of the state space explosion problem in the numerical solutions. The memory bound constraint appear in conjunction with the Markov chains dissemination to model huge complex systems. Alternative solutions soon became popular such as the structured formalisms proposed in earlier researches as powerful modeling processes. The SAN modeling formalism is based on independent components formed by states and transitions labeled by events. The available primitives for use consists of two basic types: local and synchronizing behaviors, having constant or functional rates. Such as any other structured formalism, a SAN state space is a cartesian product of subcomponents state spaces. It can produce huge representation that are practically unsolvable, even with specialized structures and state-of-the-art algorithms.

This conclusion emphasizes the advantages of each theoretical and numerical contribution as well as it shows the time-memory tradeoffs explored to reduce the impact of the state space explosion. The chapter ends enumerating open research problems and a discussion considering future works.

5.1 Thesis Summary

Considering the modeling strength and the structured representation for solving SAN models, this thesis research aims the development of new algorithms and the analysis of the internal representation of descriptors to optimize and provide exact and/or approximated solutions. Numerical algorithms are naturally chosen due to the accuracy provided in the solutions. However, they become inapplicable quickly when the size and complexity of models begins to explode. In other words, the number of synchronizing events in a model is directly related to the number of tensor products to multiply by a vector in the solution. Even storing in a memory-efficient manner, the tensor product terms multiplication can have a high computational cost. Moreover, the state space explosion imposes the

use of alternative solutions such as simulation techniques. When such techniques are applied, the research challenges are shifted to finding ways to obtain solutions approximations.

In this context, two directions were established to achieve this thesis objective: solutions when the computation time for the analytic-numeric technique is very long; and solutions when the storage requirements exceeds the memory capacity. Both problems are extremely significant in the area of stochastic modeling since there is a need for development of numerical solutions with accurate results. Also, it lacks the formal proposition of advanced alternative solutions such as discrete-event simulation based on backward coupling to provide exact samples. These issues will be properly summarized in the following sections.

5.1.1 The hybrid vector-descriptor product

The hybrid numerical solution, or the *Split* algorithm, was proposed in the Chapter 3 to reduce the computation time of a vector-descriptor product considering one iteration. In fact, the hybrid multiplication deals with the Kronecker products in a structured manner, performing matrices combinations until a *cut-parameter* σ. Actually, the non-zero values (aggregated in AUNF) related to the sparse part of *Split* are correspondent to the matrices aggregation process performed until σ, as seen in the Section 3.2 (Figure 3.3). Table 5.1 compares the vector-descriptor product methods in terms of CPU time, structure, memory and complexity (expressed in floating point multiplications). The table shows that the *Shuffle* algorithm is very memory-efficient when compared to the *Sparse* algorithm that presents a high memory demand, storing a full matrix for each tensor term. The *Split* algorithm is placed between both methods in terms of memory, balancing the computational costs through its *cut-parameter* σ.

	Sparse	Shuffle	Split
CPU time	*very efficient*	*efficient*	*very efficient*
Structure	*aggregated matrices*	*Kronecker products*	*aggregated matrices and Kronecker products*
Memory	*explosion*	*extremely efficient*	*efficient balancing the cut-parameter* σ
Complexity	$\left(\prod_{k=1}^{K} nz_k\right)$ (*Equation* 3.3)	$\left(\prod_{k=1}^{K} n_k\right) \times \left(\sum_{k=1}^{K} \frac{nz_k}{n_k}\right)$ (*Equation* 3.5)	$(\prod_{i=1}^{\sigma} nz_i) \left[\left(\prod_{i=\sigma+1}^{K} n_i\right) + \left(\prod_{i=\sigma+1}^{K} n_i \times \sum_{\substack{i=\sigma+1 \\ if f Q^{(i)} \neq Id}}^{K} \frac{nz_i}{n_i}\right)\right]$ (*Equation* 3.10)

Table 5.1: Numerical approaches comparison

The *Shuffle* method is based on the Kronecker algebra and stores only small matrices with dimensions corresponding to the automata sizes and their sparsity are correspondent to the transitions where

each event occurs. The *Sparse* method, on the contrary, considers the tensor product as a unique large matrix, aggregating these small matrices in one, through the combination of their non-zero elements in the correspondent positions. The *Split* method performs matrices aggregations to compose a sparse part to be multiplied by the structured part in the tensor format. It basically needs a definition of a *cut-parameter* σ associated to each Kronecker term, indicating the bound for matrices aggregations on its left side, and for structure maintenance on the right side.

In terms of CPU time, the *Shuffle* algorithm underperforms in time per iteration with any kind of balancing of the *cut-parameter* σ in the classical Kronecker descriptors* presented, mainly due to the shuffling process that is avoided in the sparse part when using *Split*. The extra memory spent, which is balanced according to σ, allows faster iterations and consequently less computational time for the solution of models. We consider the *Split* method as fast as the *Sparse* without loosing the storage features obtained with Kronecker-based descriptors.

Models (Section 3.2.2)	size ($\sim$Mb)	*Split* gains ($\times$)
Resource Sharing (*10_20*)	2.32	3.25
Resource Sharing (*11_14*)	3.44	3.23
Resource Sharing (*13_13*)	10.61	2.69
Resource Sharing (*14_11*)	17.89	2.64
Dining Philosophers (6)	0.02	2.00
Dining Philosophers (8)	0.24	3.03
Dining Philosophers (10)	3.94	2.67
Dining Philosophers (11)	7.21	2.13
First Available Server (12)	0.07	6.39
First Available Server (14)	1.04	8.30
First Available Server (16)	2.07	8.70
First Available Server (18)	16.51	8.21
Ad Hoc WSN (10)	0.17	5.70
Ad Hoc WSN (12)	1.43	4.64
Ad Hoc WSN (14)	13.22	5.34
Ad Hoc WSN (16)	115.34	5.36
Master-Slave (6)	3.11	3.77
Master-Slave (8)	27.57	3.70
Master-Slave (10)	234.96	3.88
Master-Slave (12)	2,229.01	3.94

Table 5.2: *Split* general performance compared with *Shuffle*

Table 5.2 presents a brief review of the *Split* algorithm gains compared to the *Shuffle* algorithm,

*The generalized Kronecker descriptors can be translated to classical Kronecker descriptors through the insertion of new synchronizing events [13].

considering some results showed in the Chapter 3 (Section 3.2.2). The column *size* is expressed in Mb representing the memory cost to run the *Split* approach. The column *gains* indicates how many times the *Split* algorithm is faster than *Shuffle* considering a complete vector-descriptor product, *i.e.* the times used to calculate the *Split* gains are the computational time spent to multiply a probability vector v by a *descriptor* $\mathcal{Q}$.

Note that for many models the memory spent in the solution is not a problem, even when we reach the total of $\sim$2.18Gb to obtain a fast result (*Master-Slave* model with 12 slaves). Additionally, one can balance the *cut-parameter* σ to spent less memory and still be faster than *Shuffle*. For all other huge models (the last variation of each class of model) the memory spent lies between 7.21Mb and 234.96Mb. Observing the gains for all examples, small and large alike, the *Split* gains in terms of computational time (after finding well-fitted *cut-parameters* for each tensor term in the descriptors) are at least twice as better than the *Shuffle* algorithm. The time gains for one iteration presented here represent an expressive overall gain in the solution with an iterative method as seen in the Section 3.3.

5.1.2 The exact simulation

The other direction developed in this thesis considers a different SAN *descriptor* representation as a discrete-event system to cope with the storage requirements when they exceed the current memory capacity. The system representation uses transition functions, running iterations for a long time until a stop criteria.

	Forward simulation	Backward simulation	Backward simulation (Monotone system)
CPU time	*fixed number of iterations*	*dependent of coupling times*	*dependent of coupling times*
Memory	*one state size*	*dependent of* $\mathcal{X}^R$ *size*	*dependent of* $\mathcal{X}^M$ *size*
Results distributed according stationary regime?	*no (bias samples)*	*yes (exact samples)*	*yes (exact samples)*
Constraints	*a valid initial state*	*valid initial states (reachable state space)*	*system must be monotone (extremal states)*
Complexity	$c_\Phi \times \tau^*$ (*Equation* 4.1)	$\lvert\mathcal{X}^R\rvert \times \mathbb{E}\tau \times c_\Phi$ (*Equation* 4.2)	$\lvert\mathcal{X}^M\rvert \times 2\mathbb{E}\tau \times c_\Phi$ (*Equation* 4.3)

Table 5.3: Simulation approaches comparison

Table 5.3 shows a summarized comparison among the simulation approaches considering CPU time, results distribution according stationary regime, constraints and complexity cost. The *Forward* simulation is included in the comparison because it is the first approach to simulate SAN specifically

focusing on the network dynamics, adapting the model state transitions as a simulation kernel [57]. Such model-driven approach was implemented starting from a pre-defined global state, running forward steps.

Section 4.1 explained the drawbacks of running forward simulations such as the problem of selecting the initial state to start the simulation, the undetermined size of the *transient* period and the consequent generation of bias samples. Note that in the context of this thesis we worked with backward coupling simulation techniques such as the *perfect sampling* because it certainly generates exact samples. Moreover, focusing in the structural aspects of models one can verify the monotonicity property and perform specialized solutions such as the monotone backward coupling simulation. The advantage of using the perfect sampling approaches is due to the fact that one can generate exact samples avoiding the stopping criteria problem [56].

The perfect sampling for SAN is an alternative for the numerical solution since it has a smaller memory demand if the model presents the monotonicity property. This is basically the reason why the structural formation is highly important. Component-wise models such as the *Queueing Network* and the *Dining Philosophers* are explored deeply in the Section 4.3.1 because both have structures that helps reducing the impact of the state space explosion. The procedure to obtain extremal initial states (Algorithm 4.5) is the first initiative in SAN towards the state space reduction based on the exploitation of component-wise state spaces.

Analyzing the results presented in the Section 4.4.2, we demonstrate expressive gains overcoming the current numerical solution limitation in terms of product state space, solving models which are too massive to run on our target machine using the current implementation of *PEPS*. The examples were extended to beyond 65 million states, *i.e.*, from 67 million to nearly 1 billion states (Table 4.2) and still our procedure was capable of producing unbiased samples for later statistical analysis.

In conclusion, simulation allows us to solve huge models and it provides an approximated result, which means that the value obtained with numerical methods are inside the corresponding simulation confidence intervals. The remaining challenge is that the simulation times are still very long mainly because there is a need of specialized structures and optimizations in the algorithms proposed which are listed in the future works section.

5.1.3 Thesis general contribution

This thesis focused on the numerical solution of the structured formalism SAN, in which the modeling simplicity and modularity brings together the state explosion problem. The automata cardinalities are combined generating a product state space $\mathcal{X}$ in which only a subset of states is reachable, forming the $\mathcal{X}^R$ set. The available numerical solutions are basically iterative solvers for Kronecker representations (sets of transition matrices), implemented in the *PEPS* software tool environment, and

a first approach to simulate SAN [57] was based on forward techniques. Considering that structured formalisms in general take advantage of their modeling characteristics as the key for performance and flexibility of their iterative solutions, the idea in this thesis was to exploit the natural models structure in hybrid and alternative numerical approaches, resulting in an efficient, or at least effective, manner to solve certain classes of huge structured models.

Figure 5.1 presents the thesis contributions graphically, showing the modeling phase evolution until the statistical analysis phase, in which the model measures can be calculated over the probability vector π generated by the numerical solutions. A model is a discrete system and the description can be expressed using sparse matrices and Kronecker operators defining a Markovian *descriptor*. A model can also be described as a table of discrete events and transition functions, defining a simulation kernel. After that, different techniques could be employed to generate the states probabilities as output. The state space explosion problem, present in both approaches, can affect also the solution phase, not only the modeling phase, and its impact is mitigated with two different perspectives.

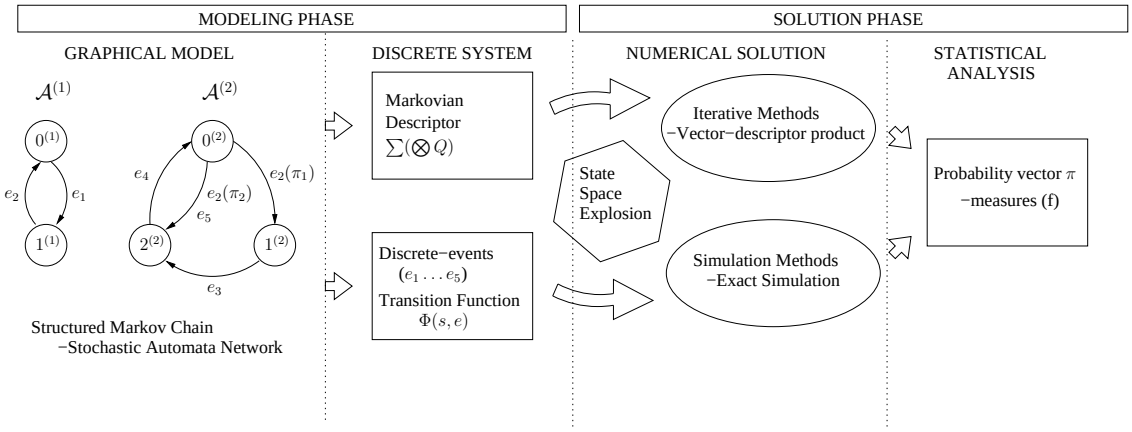


Figure 5.1: Thesis contributions scheme

It is out of the scope of this thesis to propose new storage primitives for the state space explosion problem found in the modeling phase of systems. The huge examples used in the Chapters 3 and 4 are easily extended due to the modularity and available features of the SAN formalism. Also the model reachability function can be (partially) defined in the modeling process using specific primitives.

The iterative solution is optimized through the proposition of new vector-descriptor product operations, the *Split* algorithm. The use of a Kronecker representation guarantees a memory-efficient storage, however the shuffling process involved can be computationally onerous, depending on the matrices dimensions and the number of nonzero elements (sparsity). The splitting process applied to the tensor terms aims to reduce the computational cost involved in calculating indexes to access

the vectors positions, replacing it for an extra memory needed when aggregating a subset of sparse matrices. There are many models where the synchronizations are made between few automata, *i.e.*, the matrices can be ultra-sparse[16] and aggregated in a memory-efficient way. A faster execution for large tensor terms reduces the impact of huge state spaces in the iterative approach.

The simulation solution is proposed through the adaptation of perfect sampling techniques in the context of SAN. The backward coupling procedures are not memory-efficient because we need to run as much parallel trajectories as the reachable states of the model. Due to this, the study of models with component-wise structures and partial ordering can be very useful since one can reduce the number of trajectories in parallel and obtain optimized solutions such as those provided in monotone versions. Table 5.4 generically shows a comparison among the proposed solutions to reduce the impact of state space explosion in terms of CPU time, number of iterations needed and constraints involved in each solution, considering that both can produce accurate results in different ways.

	Vector-descriptor product	Perfect sampling
CPU time	*fixed number of operations (per iteration)*	*variable coupling times (per sample generation)*
Number of iterations	*undefined until convergence (fixed precision)*	*depending of the number of samples (confidence interval)*
Results distributed according stationary regime?	*yes*	*yes*
Results accuracy	*exact solution*	*very approximated solution*
Constraints	$\mathcal{X}$ *size*	$\mathcal{X}^R$ *size or monotonicity*

Table 5.4: Numerical and simulation approaches comparison

The traditional iterative method performs a fixed number of operations (floating point multiplications) in each iteration, and iterates successively until it reaches the steady-state, in an unpredictable number of steps. It is quite different using transition functions in simulation kernels, mainly using the perfect sampling procedure, where there is no fixed number of operations to achieve the coupling of trajectories generating an exact sample (although it always happens in a finite number of steps). However, we can establish a number of steps to run (samples to collect) based on an allowed precision to obtain the result approximation. Besides, model measures can be directly calculated at each sample generation and the number of replications is dependent of the allowed precision.

In the Section 4.4.2 the Tables 4.1 and 4.2 showed that huge models are impossible to solve with the current *PEPS* software tool due to the size of $\mathcal{X}$, and the $\mathcal{X}^R$ contraction in $\mathcal{X}^M$ to run the alternative solution which is based on perfect sampling, overcoming the state space explosion. The costs in memory are also drastically reduced since for monotone versions it is sufficient the storage

of the coupling vector with extremal elements (instead of the complete reachable state space).

However, in iterative solutions, it is mandatory to store the whole state space (considering that we have no sparse implementations for the vector) at least twice to perform the traditional vector-descriptor product. In the simulation approaches, the storage of a vector of $|\mathcal{X}^R|$ positions is needed only for the statistical analysis of samples. In order to deal with this possible memory bottleneck, one can devise a method to consider only measures of interest to reject unnecessary samples. This means that monotone backward coupling methods could be more memory-efficient just improving the sampling procedure.

Concluding, the main thesis results are situated in two specific directions:

- application of classical tensor algebra properties such as the additive decomposition of tensor products, and matrices aggregation for the use of sparse techniques to accelerate the solution of Kronecker-structured models. Huge descriptors have many decomposed tensor terms, so we present a definition of a more time-efficient algorithm for the multiplication of huge vectors by complex structures based on Kronecker operations among sparse matrices (called vector-descriptor product);

- application of advanced simulation techniques based on backward coupling for complex structures such as those provided by SAN, reducing the impact of the state space explosion problem which can avoid the use of iterative solutions. It is analyzed a component-wise structure to take advantage of the monotonicity property of events for state space contraction, and then a consequent memory-efficient backward simulation is provided.

5.2 Open Problems and Future Works

The analysis of large Markovian models suffers from state space explosion and future researches may focus on the limiting factors for solutions which are basically memory and time. Apart from using sophisticated computational representations for a compact descriptor storage, the SAN formalism demands new efficient analysis techniques exploiting the models structure. Due to this it is natural to take advantage of its power of modeling (functional primitives and synchronized interaction of disjoint components) also in the solutions.

The *Split* algorithm is proposed to classical Kronecker descriptors so the next step is the proposition of a generalized version which can solve matrices with functional rates instead of only constant ones. A similar work about these functional dependencies inside tensor terms changed completely

the performance of the *Shuffle* algorithm [37]. One can estimate that similar gains with functional dependencies analysis (and possible automata permutations) could benefit the *Split* algorithm as well. However, the main aspect related to its performance is derived from the flexibility proposed in the treatment of the structured representation as seen in the Chapter 3. A clearly open problem is the choice of the division point in each tensor product term (choice of the cut-parameter σ) and, even more important, the choice of matrices permutations in the terms providing a more efficient aggregation at left and few multiplications at right.

The research for an heuristic to automatically choose a satisfactory permutation of matrices and the *cut-parameter*, for each tensor product, is a considerable challenge. This is not a trivial task, due to the tensor product term formation and intrinsic matrices details such as dimensions, total of nonzero elements and computational cost in multiplications. These parameters open the possibility of a thorough analysis of the related theoretical computational cost to obtain a generalized version for the *Split* algorithm, with a finite set of rules for the proper definition of the *cut-parameter* based on matrices properties and permutations.

This thesis also introduced an advanced simulation technique (*perfect sampling*) to the context of SAN using its underlying system dynamic expressed by different events in a *descriptor*. A future work in this direction is the development of optimized procedures to find extremal states in general chains, not only in those underlying component-wise models as presented in the Chapter 4. Also theoretical improvements are needed for a deep understanding of the transition function Φ properties and the bounds on the coupling time τ.

There are researches conducted towards to the improvement of the simulation complexity using variance reduction techniques and functional coupling [65] to accelerate the samples generation. Preliminary results shows that for the component-wise models in which the numerical solution is no longer possible with PEPS, perfect sampling techniques are a reasonable alternative. A qualitative analysis of SAN through new tools to facilitate the comprehension of complex interactions among automata could provide benefits from the structural aspects.

Additionally, since simulation uses statistical techniques to analyze output data, and structured analytical models can have thousands of reachable states collected during these experiments, a functional analysis can help maintaining the memory resources to manageable limits, using these methods based on the coupling of parallel trajectories. Avoiding the storage of a huge vector is possible considering specific of measures of interest as parameters for the sampling procedure.

The numerical methods proposed for the SAN formalism in this thesis, beyond their inherent complexity, could be enhanced using parallel implementations since both algorithms present a natural independence of operations. The *Split* algorithm allows independence among normal factors due to the additive decomposition property exploited. The tensor products can be analyzed separately

without the need to share the probability vector during multiplication. The *perfect sampling* generates independent and exact samples so one can also run independent batches of experiments without any solution bias. An interesting aspect for future research in parallelization should consider the amount of memory needed, the processing demands (the number of floating point multiplications or transition function operations), volume of data exchanged (size of vectors) to be as evenly as possible distributed among parallel machines.

In the future we hope that the development and optimizations of these approaches may, virtually, have no size bound, since neither the transition matrix, nor the probability vector would need to be stored as a single matrix or vector. Such possible applications will only have to deal with the time bound optimization, then the solution of models with thousands of million states will become as usual as our current tens of millions bound.

Bibliography

[1] M. Ajmone-Marsan, G. Conte, and G. Balbo. A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. *ACM Transactions on Computer Systems*, 2(2):93–122, 1984.

[2] V. Amoia, G. De Micheli, and M. Santomauro. Computer-Oriented Formulation of Transition-Rate Matrices via Kronecker Algebra. *IEEE Transactions on Reliability*, R-30(2):123–132, 1981.

[3] L. Baldo, L. Brenner, L. G. Fernandes, P. Fernandes, and A. Sales. Performance Models for Master/Slave Parallel Programs. *Electronic Notes In Theoretical Computer Science*, 128(4):101–121, April 2005.

[4] L. Baldo, L. G. Fernandes, P. Roisenberg, P. Velho, and T. Webber. Parallel PEPS Tool Performance Analysis using Stochastic Automata Networks. In M. Donelutto, D. Laforenza, and M. Vanneschi, editors, *International Conference on Parallel Processing (Euro-Par 2004)*, volume 3149 of *LNCS*, pages 214–219, Berlin, Germany, December 2004. Springer-Verlag Heidelberg.

[5] A. Benoit, L. Brenner, P. Fernandes, B. Plateau, and W. J. Stewart. The PEPS Software Tool. In *Computer Performance Evaluation (TOOLS 2003)*, volume 2794 of *LNCS*, pages 98–115. Springer-Verlag Heidelberg, 2003.

[6] A. Benoit, P. Fernandes, B. Plateau, and W. J. Stewart. On the benefits of using functional transitions and Kronecker algebra. *Performance Evaluation*, 58(4):367–390, December 2004.

[7] A. Benoit, B. Plateau, and W. J. Stewart. Memory-efficient Kronecker algorithms with applications to the modelling of parallel systems. *Future Generation Computer Systems*, 22(7):838–847, 2004.

[8] C. Bertolini, L. Brenner, P. Fernandes, A. Sales, and A. F. Zorzo. Structured Stochastic Modeling of Fault-Tolerant Systems. In *Proceedings of the 12th IEEE/ACM Internacional Symposium*

on Modelling, Analysis and Simulation on Computer and Telecommunication Systems (MASCOTS'04), pages 139–146, Volendam, The Netherlands, October 2004. IEEE Press.

[9] A. A. Borovkov and S. G. Foss. Two ergodicity criteria for stochastically recursive sequences. *Journal Acta Applicandae Mathematicae: An International Survey Journal on Applying Mathematics and Mathematical Applications*, 34:125–134, February 1994.

[10] A. Bouillard and B. Gaujal. Backward coupling in petri nets. In *Proceedings of the 1st international conference on Performance evaluation methodologies and tools (Valuetools'06)*, page 33, New York, NY, USA, 2006. ACM Press.

[11] A. Bouillard and B. Gaujal. Backward Coupling in Bounded Free-Choice Nets Under Markovian and Non-Markovian Assumptions. *Discrete Event Dynamic Systems: Theory and Applications*, 18(4):473–498, December 2008.

[12] L. Brenner, P. Fernandes, J. M. Fourneau, and B. Plateau. Modelling Grid5000 point availability with SAN. In *Proceedings of the Third International Workshop on Practical Applications of Stochastic Modelling (PASM'08)*, pages 149–162, 2008.

[13] L. Brenner, P. Fernandes, and A. Sales. The Need for and the Advantages of Generalized Tensor Algebra for Kronecker Structured Representations. *International Journal of Simulation: Systems, Science & Technology (IJSIM)*, 6(3-4):52–60, February 2005.

[14] P. Buchholz. A distributed numerical/simulative algorithm for the analysis of large continuous time Markov chains. In *Proceedings of the eleventh Workshop on Parallel and Distributed Simulation (PADS'97)*, pages 4–11, Washington, DC, USA, 1997. IEEE Computer Society.

[15] P. Buchholz. A new approach combining simulation and randomization for the analysis of large continuous time Markov Chains. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(2):194–222, 1998.

[16] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS Journal on Computing*, 12(3):203–222, July 2000.

[17] P. Buchholz and P. Kemper. Hierarchical reachability graph generation for Petri nets. *Formal Methods in Systems Design*, 21(3):281–315, 2002.

[18] A. Busic, B. Gaujal, and J. M. Vincent. Perfect simulation and non-monotone Markovian systems. In *Proceedings of the 3rd international Conference on Performance Evaluation Methodologies and Tools (ValueTools'08)*, pages 1–10, Athens, Greece, October 2008.

[19] Y. Cai. A non-monotone CFTP perfect simulation method. *Statistica Sinica*, 15(4):927–943, 2005.

[20] R. Chanin, M. Corrêa, P. Fernandes, A. Sales, R. Scheer, and A. F. Zorzo. Analytical Modeling for Operating System Schedulers on NUMA Systems. In *Proceedings of the Second International Workshop on the Practical Application of Stochastic Modeling (PASM 2005)*, volume 151 of *Electronic Notes in Theoretical Computer Science*, pages 131–149, June 2006.

[21] G. Ciardo, M. Forno, P. L. E. Grieco, and A. S. Miner. Comparing implicit representations of large CTMCs. In *Proceedings of the 4th International Conference on the Numerical Solution of Markov Chains (NSMC 2003)*, pages 323–327, September 2003.

[22] G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. SMART: Stochastic Model Analyzer for Reliability and Timing. In *Tools of Aachen 2001 International Multiconference on Measurement, Modelling and Evaluation of Computer-Communication Systems*, pages 29–34, September 2001.

[23] G. Ciardo and A. S. Miner. Storage Alternatives for Large Structured State Spaces. In *Proceedings of the 9th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, volume 1245 of *LNCS*, pages 44–57. Springer-Verlag Heidelberg, 1997.

[24] R. M. Czekster, P. Fernandes, J.-M. Vincent, and T. Webber. Split: a flexible and efficient algorithm to vector-descriptor product. In *Proceedings of the 2nd international conference on Performance evaluation methodologies and tools (ValueTools'07)*, volume 321 of *ACM International Conference Proceeding Series*, Brussels, Belgium, Belgium, 2007. Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (ICST).

[25] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 2nd edition, 2002.

[26] M. Davio. Kronecker Products and Shuffle Algebra. *IEEE Transactions on Computers*, 30(2):116–125, February 1981.

[27] D. D. Deavours and W. H. Sanders. An Efficient Disk-Based Tool for Solving Very Large Markov Models. In *Proceedings of the 9th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools*, volume 1245 of *LNCS*, pages 58–71, 1997.

[28] D. D. Deavours and W. H. Sanders. On-the-fly Solution Techniques for Stochastic Petri Nets and Extensions. In *Proceedings of the 6th International Workshop on Petri Nets and Performance Models (PNPM'97)*, pages 132–141, Washington, DC, USA, 1997. IEEE Computer Society.

[29] F. Delamare, F. L. Dotti, P. Fernandes, C. M. Nunes, and L. C. Ost. Analytical modeling of random waypoint mobility patterns. In *Proceedings of the 3rd ACM international workshop on Performance evaluation of wireless ad hoc, sensor and ubiquitous networks (PE-WASUN'06)*, pages 106–113, New York, NY, USA, 2006. ACM Press.

[30] X. K. Dimakos. A Guide to Exact Simulation. *International Statistical Review*, 69(1):27–48, 2001.

[31] S. Donatelli. Superposed stochastic automata: a class of stochastic Petri nets with parallel solution and distributed state space. *Performance Evaluation*, 18(1):21–36, July 1993.

[32] S. Donatelli. Superposed generalized stochastic Petri nets: definition and efficient solution. In R. Valette, editor, *Proceedings of the 15th International Conference on Applications and Theory of Petri Nets*, pages 258–277. Springer-Verlag Heidelberg, 1994.

[33] S. Donatelli. Kronecker Algebra and (Stochastic) Petri Nets: Is It Worth the Effort? In J. M. Colom & M. Koutny, editor, *Proceedings of the 22nd International Conference on Applications and Theory of Petri Nets*, volume 2075 of *LNCS*, pages 1–18, London, UK, 2001. Springer-Verlag Heidelberg.

[34] F. L. Dotti, P. Fernandes, A. Sales, and O. M. Santos. Modular Analytical Performance Models for Ad Hoc Wireless Networks. In *Proceedings of the Third International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt'05)*, pages 164–173, Washington, DC, USA, April 2005. IEEE Computer Society.

[35] A. G. Farina, P. Fernandes, and F. M. Oliveira. Representing software usage models with Stochastic Automata Networks. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 401–407. ACM Press, 2002.

[36] P. Fernandes and B. Plateau. Modeling Finite Capacity Queueing Networks with Stochastic Automata Networks. In *Proceedings of the 4th International Workshop on Queueing Networks with Finite Capacity (QNETs 2000)*, pages 1–12, July 2000.

[37] P. Fernandes, B. Plateau, and W. J. Stewart. Efficient descriptor-vector multiplication in Stochastic Automata Networks. *Journal of the ACM SIGMETRICS (JACM)*, 45(3):381–414, May 1998.

[38] P. Fernandes, J. M. Vincent, and T. Webber. Perfect Simulation of Stochastic Automata Networks. In *Proceedings of 15th International Conference on Analytical and Stochastic Modelling Techniques and Applications (ASMTA'08)*, volume 5055 of *LNCS*, pages 249–263. Springer-Verlag Heidelberg, June 2008.

[39] J. A. Fill and M. Machida. Stochastic Monotonicity and Realizable Monotonicity. *Annals of Probability*, 29(2):938–978, 2001.

[40] J. M. Fourneau, I. Kadi, N. Pekergin, J. Vienne, and J. M. Vincent. Perfect simulation and monotone stochastic bounds. In *Proceedings of the 2nd International Conference on Performance Evaluation Methodolgies and Tools (VALUETOOLS'07)*, volume 321 of *ACM International Conference Proceeding Series*, pages 65–73, Brussels, Belgium, Belgium, 2007. Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (ICST).

[41] S. Gilmore and J. Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Proceedings of the 7th international conference on Computer Performance Evaluation : modelling techniques and tools*, pages 353–368, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.

[42] P. Glasserman and D. D. Yao. *Monotone structure in discrete-event systems*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[43] O. Häggström. *Finite Markov Chains and Algorithmic Applications*. Cambridge University Press, Cambridge, UK, 2002.

[44] J. Hillston. *A compositional approach to performance modelling*. Cambridge University Press, New York, USA, 1996.

[45] J. Hillston and L. Kloul. An Efficient Kronecker Representation for PEPA models. In L. de Alfaro and S. Gilmore, editors, *Proceedings of the First joint PAPM-PROBMIV Workshop)*, pages 120–135. Springer-Verlag Heidelberg, September 2001.

[46] V. V. Lam, P. Buchholz, and W. H. Sanders. A Structured Path-Based Approach for Computing Transient Rewards of Large CTMCs. In *Proceedings of the The Quantitative Evaluation of Systems, First International Conference on (QEST'04)*, pages 136–145, Washington, DC, USA, 2004. IEEE Computer Society.

[47] V. V. Lam, P. Buchholz, and W. H. Sanders. A component-level path-based simulation approach for efficient analysis of large Markov models. In *Proceedings of the 37th conference on Winter simulation (WSC'05)*, pages 584–590. Winter Simulation Conference, 2005.

[48] A.M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. MacGraw-Hill, New York, USA, 1991.

[49] L.Brenner, P.Fernandes, B.Plateau, and I.Sbeity. PEPS2007 - Stochastic Automata Networks Software Tool. In *Proceedings of the 4th International Conference on Quantitative Evaluation of Systems (QEST 2007)*, pages 163–164. IEEE Press, 2007.

[50] J. Li, C. Blake, D. S. J. De Couto, H. I. Lee, and R. Morris. Capacity of Ad Hoc Wireless Networks. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, pages 61–69. ACM Press, July 2001.

[51] T. Lindvall. *Lectures on the Coupling Method*. John Wiley & Sons, Inc., New York, USA, 1992.

[52] A. S. Miner and G. Ciardo. Efficient Reachability Set Generation and Storage Using Decision Diagrams. In *Proceedings of the 20th International Conference on Applications and Theory of Petri Nets (ICATPN'99)*, volume 1639 of *LNCS*, pages 6–25, Williamsburg, VA, USA, June 1999. Springer-Verlag Heidelberg.

[53] L. Mokdad, J. Ben-Othman, and A. Gueroui. Quality of Service of a Rerouting Algorithm Using Stochastic Automata Networks. In *Proceedings of the 6th IEEE Symposium on Computers and Communications*, pages 338–343. IEEE Computer Society, July 2001.

[54] B. Plateau. On the stochastic structure of parallelism and synchronization models for distributed algorithms. *ACM SIGMETRICS Performance Evaluation Review*, 13(2):147–154, August 1985.

[55] B. Plateau and K. Atif. Stochastic Automata Networks for modelling parallel systems. *IEEE Transactions on Software Engineering*, 17(10):1093–1108, October 1991.

[56] J. G. Propp and D. B. Wilson. Exact Sampling with Coupled Markov Chains and Applications to Statistical Mechanics. *Random Structures and Algorithms*, 9(1–2):223–252, 1996.

[57] W. J. Stewart R. Jungblut-Hessel, B. Plateau and B. Ycart. Fast simulation for Road Traffic Network. *RAIRO Operational Research*, 35(2):229–250, June 2001.

[58] S. M. Ross. *Simulation*. Academic Press, Inc., Orlando, FL, USA, 2002.

[59] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, Boston, MA, USA, 1995.

[60] Ö. Stenflo. Ergodic Theorems for Markov chains represented by Iterated Function Systems. *Bulletin of the Polish Academy of Sciences, Mathematics*, 49(1):27–43, 2001.

[61] W. J. Stewart. *Introduction to the numerical solution of Markov chains*. Princeton University Press, Princeton, NJ, USA, 1994.

[62] C. Tadonki and B. Philippe. Parallel Multiplication of a Vector by a Kronecker Tensor Product of matrices. *Parallel numerical linear algebra*, pages 71–89, 2000.

[63] J.-M. Vincent. Perfect simulation of monotone systems for rare event probability estimation. In *Proceedings of the 37th Conference on Winter Simulation*, pages 528–537. Winter Simulation Conference, 2005.

[64] J.-M. Vincent. Perfect Simulation of Queueing Networks with Blocking and Rejection. In *Proceedings of the 2005 Symposium on Applications and the Internet Workshops*, pages 268–271, Washington, DC, USA, 2005. IEEE Computer Society.

[65] J.-M. Vincent and C. Marchand. On the exact simulation of functionals of stationary Markov chains. *Linear Algebra and its Applications*, 386:285–310, 2004.

[66] J.-M. Vincent and J. Vienne. Perfect simulation of index based routing queueing networks. *ACM SIGMETRICS Performance Evaluation Review*, 34(2):24–25, 2006.

Appendix A

SAN Examples

This appendix presents the following SAN models descriptions used as examples for numerical and theoretical results. Their graphical representations are also included here.

- a simple queueing network model (A.1);

- a resource sharing model with a pool of resources (A.2);

- the *dining philosophers* model in two versions (with and without resource reservation) (A.3);

- a model to analyze servers availability (A.4);

- a model to analyze an ad hoc wireless sensor network (A.5);

- a model to analyze a parallel implementation (A.6).

A.1 *Queueing Network* model

We introduce this section with a queueing system conversion in a SAN model [36] whose the translation is trivial considering the interaction among queues and the independent behavior of client arrivals and some departures. Figure A.1 represents a simple queueing network with two queues of capacities given by K_i, the arrival rate in the system is α_1, the routing rate between queues is α_2 (with loss), and departure rate of the second queue is α_3.

The equivalent SAN model has two automata $A^{(1)}$ and $A^{(2)}$ representing both queues respectively, and three events composing the set ξ (since e_1 and e_2 are local events, and e_{12} is a synchronizing event between automata) with their constant rates. In general, each queue can be represented by an automaton $A^{(i)}$ composed of $K_i + 1$ states representing the number of clients in the queue (*i.e.*, the

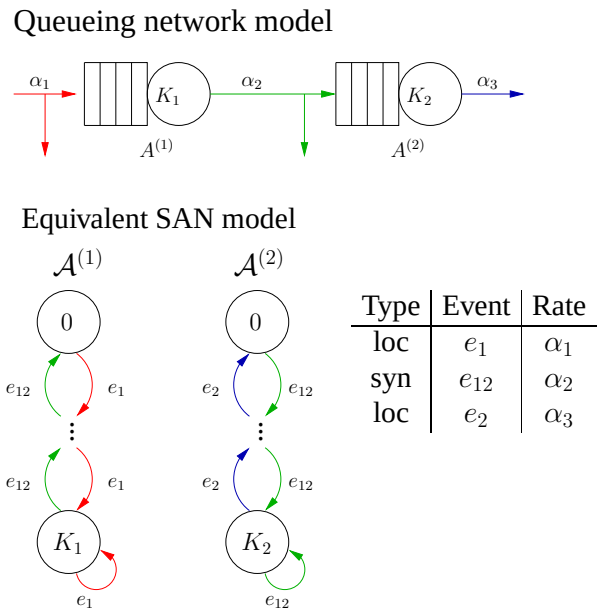


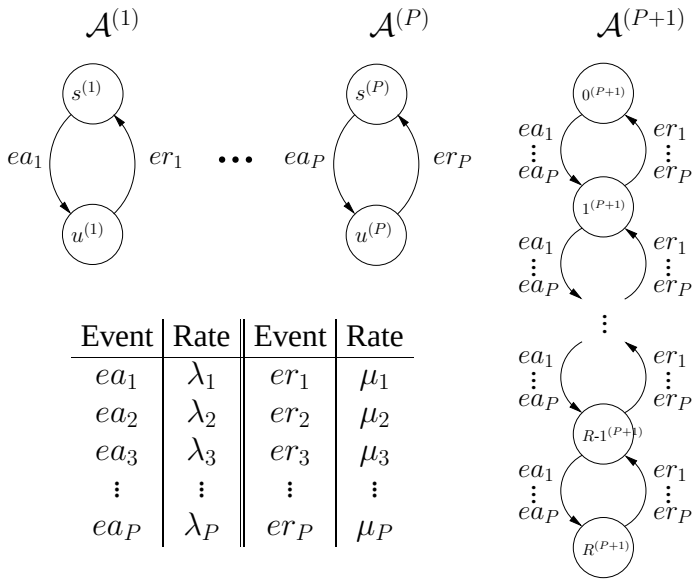
Figure A.1: Queueing network and equivalent SAN model

state 0 represents that a queue is empty, and the states K_1 or K_2 represent that the queue is full). The product state space $\mathcal{X}$ of this model is formed by $(K_1 + 1) \times (K_2 + 1)$ global states. All states are reachable in this model. Despite of that, one can define a partial reachability indicating, for example, that both queues are empty: $\mathcal{F}^{R*} = (\text{st } A^{(1)} == 0^{(1)})$ && $(\text{st } A^{(2)} == 0^{(2)})$;

A.2 *Resource Sharing* model

Figure A.2 represents a classical resource sharing system with P processes sharing R resources. Each process is represented by an automaton $A^{(i)}$ ($i = 1 \ldots P$) composed of two states: $s^{(i)}$ (sleeping) and $u^{(i)}$ (using). A resource pool is represented by the automaton $A^{(P+1)}$ and it has $R + 1$ states indicating the number of resources in use.

The model presents only synchronizing events composing the set ξ, since the events ea_i represent the acquiring of a resource with constant rate λ_i, and the events er_i represent the release of a resource with constant rate μ_i. The product state space $\mathcal{X}$ of this model is formed by $2^P \times (R+1)$ global states. One can define a partial reachability function indicating, for example, that the number of automata in the sleeping state $s^{(i)}$ is equal to P, *i.e.* all processes are sleeping: $\mathcal{F}^{R*} = (\text{nb } A^{(i)} [\, s^{(i)} \,]) == P$;



Event	Rate	Event	Rate
ea_1	λ_1	er_1	μ_1
ea_2	λ_2	er_2	μ_2
ea_3	λ_3	er_3	μ_3
$\vdots$	$\vdots$	$\vdots$	$\vdots$
ea_P	λ_P	er_P	μ_P

Figure A.2: Classical resource sharing SAN model

A.3 *Dining Philosophers* model

This section presents another classical performance model to analyze mutual exclusion in resource sharing. The modeling abstraction is called the *dining philosophers problem* and is summarized as K philosophers sitting at a table doing one of two things - eating or thinking.

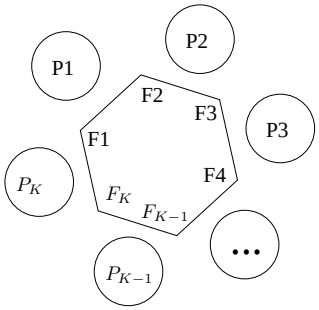
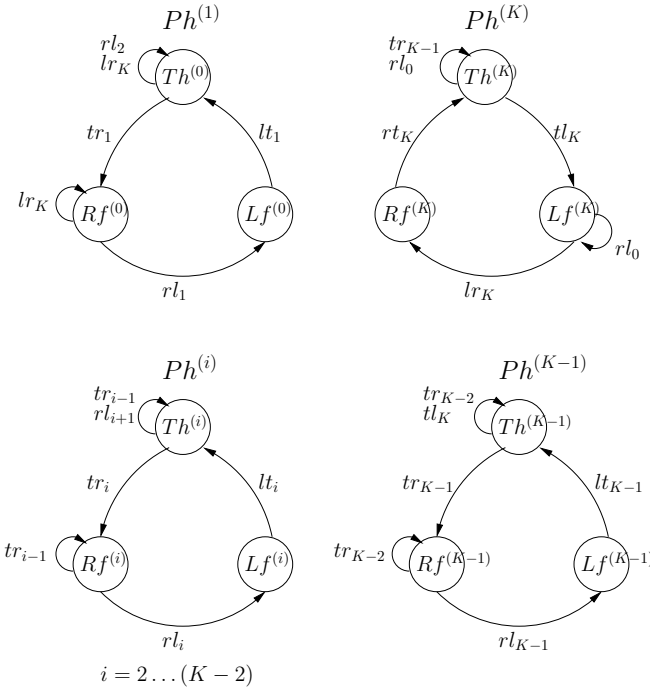


Figure A.3: Dining Philosophers table configuration

The philosophers sit at a circular table (Figure A.3) with a large bowl of food in the center. A fork F_k is placed between each philosopher P_k, and as such, each philosopher has one fork to his left and one fork to his right. The philosopher must have two forks (at the same time) to eat.

A.3.1 *Dining Philosophers* model (with resource reservation)

The SAN model in Figure A.4 has K automata $Ph^{(k)}$ representing the philosophers, each one with three states: $Th^{(k)}$ (thinking), $Lf^{(k)}$ (taking left fork), $Rf^{(k)}$ (taking right fork). The model allows one fork reservation to after acquiring the second fork.



Type	Event	Rate
loc	lt_i	μ_i
syn	tr_i	λ_i
syn	rl_i	λ_i
loc	rt_K	μ_K
syn	tl_K	λ_K
syn	lr_K	λ_K

Figure A.4: Dining Philosophers SAN model with reservation

The philosopher can reserve the fork on his immediate left or right waiting for eating with two available forks. To avoid deadlock is established an ordering to get the forks in the table, for each philosopher in the model. Then the model presents synchronizing events with constant rates for taking the right and left forks (tr_i, rl_i, tl_K and lr_K) and local events (lt_i and rt_K) representing the release of forks. The product state space $\mathcal{X}$ of this model is formed by 3^K states. The partial reachability function can be defined, for example, indicating that all philosophers are thinking, so the number of automata in the thinking state $Th^{(k)}$ is equal to K: $\mathcal{F}^{R*} = (\text{nb } Ph^{(k)} \, [\, Th^{(k)} \,]) \; == \; K$;

A.3.2 *Dining Philosophers* model (without resource reservation)

The SAN model in Figure A.5 has K automata $P^{(k)}$ representing the philosophers, each one with two states: $T^{(k)}$ (thinking) and $E^{(k)}$ (eating). The state $E^{(k)}$ supposes the philosopher required both forks at the same time to eat, without reserving earlier the first fork to then take the second.

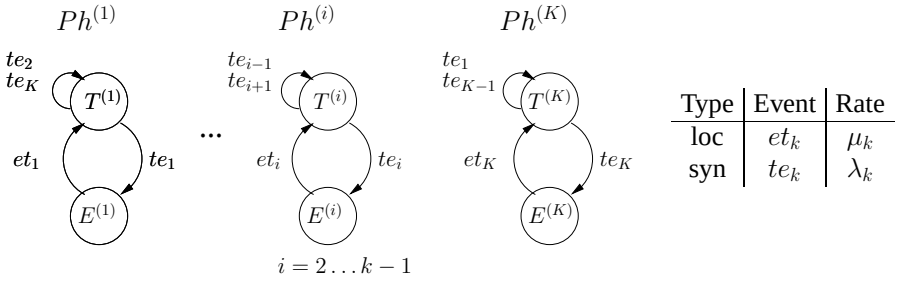


Figure A.5: Dining Philosophers SAN model without reservation

Then the model presents synchronizing events with constant rates for eating (te_K) and local events (et_K) representing the release of forks returning to thinking. The product state space $\mathcal{X}$ of this model is formed by 2^K states. The partial reachability function can be defined, for example, indicating that all philosophers are thinking, so the number of automata in the thinking state $T^{(k)}$ is equal to K: $\mathcal{F}^{R*} = (\text{nb } Ph^{(k)} [\, T^{(k)} \,]) == K$;

A.4 *First Available Server* model

This section presents a model to analyze server availability considering N servers. Each server $A^{(i)}$ has two states: $I^{(i)}$ (idle) and $B^{(i)}$ (busy). In this example, packages arriving at a *servers switch block*, depart through the first output port (or server) that is not busy, as long as at least one server is not blocked.

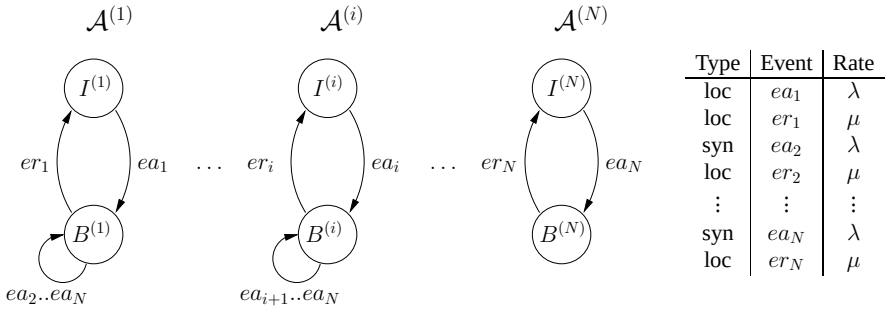


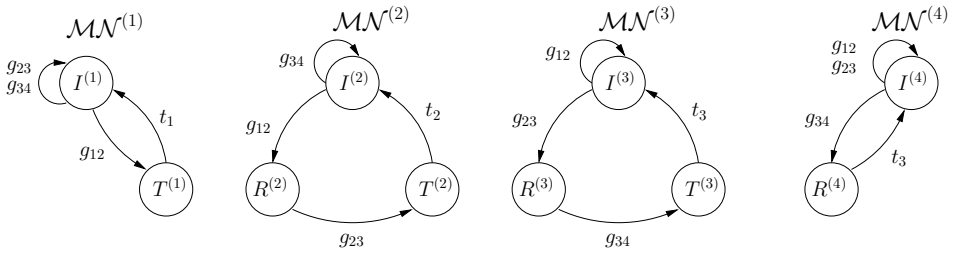
Figure A.6: First available server SAN model

The SAN model in Figure A.6 can be viewed as a framework for analysis of different queueing systems (*e.g.*call centers lines occupation). Each package in the queue can advance as soon as possible to the first available server without preferring one over another (*i.e.*, the priority of servers is given by themselves).

The model has synchronizing events ea_i ($i = 2 \ldots N$) turning the servers busy. The local events are: ea_1 (package arrival) and er_i (to turn the servers idle). All events present constant rates. The product state space $\mathcal{X}$ of this model is formed by 2^N states. The partial reachability function can be defined, for example, indicating that all servers are idle: $\mathcal{F}^{R*} = (\text{nb } A^{(i)} \ [\ I^{(i)}\ ])\ ==\ N$;

A.5 *Ad Hoc Wireless Sensor Network* model

The SAN model in the Figure A.7 represents a chain of four mobile nodes in a Wireless Sensor Network (Ad Hoc WSN model) running over the 802.11 standard for ad hoc networks. This model [34] resembles the ad hoc forwarding experiment presented in [50] using SAN. The chain can be generically modeled with N nodes, where the first node $\mathcal{MN}^{(1)}$ (*Source* automaton) generates the packets as fast as the standard allows. The packets are forwarded through the chain by the *Relay* automata called $\mathcal{MN}^{(i)}$, where the variable i is among the value 2 and $(N-1)$, until the last node $\mathcal{MN}^{(N)}$ (*Sink* automaton).



Type	Event	Rate	Type	Event	Rate
loc	t_1	μ_1	syn	g_{12}	λ_{12}
loc	t_2	μ_2	syn	g_{23}	λ_{23}
syn	t_3	μ_3	syn	g_{34}	λ_{34}

Figure A.7: Ad hoc wireless sensor network SAN model (4 nodes)

Generically, the model has local events t_i ($i = 1 \ldots N-2$) and a synchronizing event t_{N-1} representing the end of the packets transmission. The synchronizing events g_{12}, g_{23} and g_{34} are activating the packets forwarding process. All events present constant rates. The product state space $\mathcal{X}$ of this model is formed by $2^2 \times 3^{N-2}$ states. The partial reachability function can be defined indicating that the *Source* automaton $\mathcal{MN}^{(1)}$ is in the idle state: $\mathcal{F}^{R*} = (\text{st } \mathcal{MN}^{(1)}\ ==\ I^{(1)})$;

A.6 *Master-Slave Parallel Algorithm* model

Figure A.8 refers to an evaluation of the master-slave parallel implementation of the Propagation algorithm considering asynchronous communication [3], indicating to parallel program developers what are the possible execution bottlenecks before the implementation. This SAN model contains one *Master* automaton, one huge *Buffer* automaton, and N automata $Slave^{(i)}$, where $i = 1 \ldots N$. The *Master* automaton presents three states: Tx (transmitting), Rx (receiving) and ITx (idle). The *Buffer* automaton has K positions (states) plus an empty state 0. The $Slave^{(i)}$ automata presents three states: I (idle), Pr (processing) and Tx (transmitting).

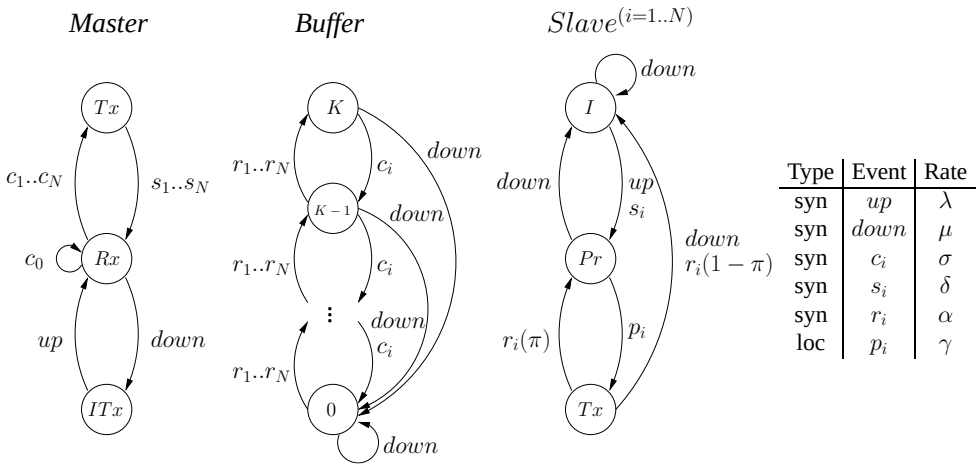


Figure A.8: Master-slave parallel algorithm SAN model

The model has synchronizing events related to the *Master* and $Slave^{(i)}$ activities controlling also the *Buffer*: up, $down$, c_i and s_i. The *Buffer* is accessed by the slaves using synchronizing event r_i. The *Master* also fills the buffer with the synchronizing event c_i. The slaves start transmissions with a local event p_i. This model can vary defining different numbers of slaves and sizes for the buffer. The product state space $\mathcal{X}$ of this model is given by $3^{N+1} \times (K+1)$ states. The partial reachability function can be defined indicating that the *Master* automaton is in the idle state: $\mathcal{F}^{R*} = (\text{st}\ Master\ ==\ ITx)$;

Appendix B

Kronecker Algebra

Kronecker (tensor) algebra is an algebra defined on matrices with a product operator and a sum operator [2, 26]. The classical tensor algebra (CTA) consider that the matrices elements are constant values. This appendix addresses properties and characteristics of these matrix operators ($\bigotimes$ and $\bigoplus$).

B.1 Kronecker (tensor) product

In general, to define the tensor product of two matrices, $\mathcal{Q}^{(1)}$ of dimensions $(\rho_1 \times \gamma_1)$ and $\mathcal{Q}^{(2)}$ of dimensions $(\rho_2 \times \gamma_2)$, we have $\mathcal{Q} = (\mathcal{Q}^{(1)} \otimes \mathcal{Q}^{(2)})$ as a matrix with dimensions $(\rho_1\rho_2 \times \gamma_1\gamma_2)$.

However, the tensor $\mathcal{Q}$ is a four dimension tensor, which can be flattened (*i.e.* put in a two-dimension format) in a single matrix $\mathcal{Q}$ consisting of $\rho_1\gamma_1$ blocks each having dimensions $(\rho_2\gamma_2)$. To specify a particular element, it suffices to specify the block in which the element occurs and the position within that block of the element under consideration. Thus, the matrix $\mathcal{Q}$ element q_{36} (which corresponds to tensor $\mathcal{Q}$ element $\mathcal{Q}_{[1,0][1,2]}$ is in the $(1, 1)$ block and at position $(0, 2)$ of that block and has the numeric value $q_{11}^{(1)}; q_{02}^{(2)}$). Algebraically, the tensor $\mathcal{Q}$ elements are defined by:

$$q_{[ik][jl]} = q_{ij}^{(1)} q_{kl}^{(2)}$$

Defining two matrices $\mathcal{Q}^{(1)}$ and $\mathcal{Q}^{(2)}$ as follows:

$$\mathcal{Q}^{(1)} = \begin{pmatrix} q_{00}^{(1)} & q_{01}^{(1)} & q_{02}^{(1)} & q_{03}^{(1)} \\ q_{10}^{(1)} & q_{11}^{(1)} & q_{12}^{(1)} & q_{13}^{(1)} \\ q_{20}^{(1)} & q_{21}^{(1)} & q_{22}^{(1)} & q_{23}^{(1)} \end{pmatrix} \qquad \mathcal{Q}^{(2)} = \begin{pmatrix} q_{00}^{(2)} & q_{01}^{(2)} \\ q_{10}^{(2)} & q_{11}^{(2)} \end{pmatrix}$$

The *tensor product* $\mathcal{Q} = \mathcal{Q}^{(1)} \otimes \mathcal{Q}^{(2)}$ is therefore given by

$$\mathcal{Q} = \left(\begin{array}{c|c|c|c} q_{00}^{(1)}\mathcal{Q}^{(2)} & q_{01}^{(1)}\mathcal{Q}^{(2)} & q_{02}^{(1)}\mathcal{Q}^{(2)} & q_{03}^{(1)}\mathcal{Q}^{(2)} \\ \hline q_{10}^{(1)}\mathcal{Q}^{(2)} & q_{11}^{(1)}\mathcal{Q}^{(2)} & q_{12}^{(1)}\mathcal{Q}^{(2)} & q_{13}^{(1)}\mathcal{Q}^{(2)} \\ \hline q_{20}^{(1)}\mathcal{Q}^{(2)} & q_{21}^{(1)}\mathcal{Q}^{(2)} & q_{22}^{(1)}\mathcal{Q}^{(2)} & q_{23}^{(1)}\mathcal{Q}^{(2)} \end{array}\right)$$

A particularly important type of tensor product is the tensor product where one of the matrices is an identity matrix of order n (I_n). These particular tensor products are called *normal factors* and they can be composed by matrices only on diagonal blocks:

$$I_3 \otimes \mathcal{Q}^{(2)} = \left(\begin{array}{cc|cc|cc} q_{00}^{(2)} & q_{01}^{(2)} & 0 & 0 & 0 & 0 \\ q_{10}^{(2)} & q_{11}^{(2)} & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & q_{00}^{(2)} & q_{01}^{(2)} & 0 & 0 \\ 0 & 0 & q_{10}^{(2)} & q_{11}^{(2)} & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & q_{00}^{(2)} & q_{01}^{(2)} \\ 0 & 0 & 0 & 0 & q_{10}^{(2)} & q_{11}^{(2)} \end{array}\right)$$

or diagonal matrices in every block:

$$\mathcal{Q}^{(2)} \otimes I_3 = \left(\begin{array}{ccc|ccc} q_{00}^{(2)} & 0 & 0 & q_{01}^{(2)} & 0 & 0 \\ 0 & q_{00}^{(2)} & 0 & 0 & q_{01}^{(2)} & 0 \\ 0 & 0 & q_{00}^{(2)} & 0 & 0 & q_{01}^{(2)} \\ \hline q_{10}^{(2)} & 0 & 0 & q_{11}^{(2)} & 0 & 0 \\ 0 & q_{10}^{(2)} & 0 & 0 & q_{11}^{(2)} & 0 \\ 0 & 0 & q_{10}^{(2)} & 0 & 0 & q_{11}^{(2)} \end{array}\right)$$

B.2 Kronecker (tensor) sum

The *tensor sum* of two square matrices $\mathcal{Q}^{(1)}$ and $\mathcal{Q}^{(2)}$ is defined in terms of tensor products as the sum of normal factors of matrices $\mathcal{Q}^{(1)}$ and $\mathcal{Q}^{(2)}$, *i.e.*:

$$\mathcal{Q}^{(1)} \oplus \mathcal{Q}^{(2)} = \mathcal{Q}^{(1)} \otimes I_{n_{\mathcal{Q}^{(2)}}} + I_{n_{\mathcal{Q}^{(1)}}} \otimes \mathcal{Q}^{(2)}$$

where $n_{\mathcal{Q}^{(1)}}$ and $n_{\mathcal{Q}^{(2)}}$ are respectively the orders of the matrices $\mathcal{Q}^{(1)}$ and $\mathcal{Q}^{(2)}$. Since both sides of the usual matrix addition operation must have identical dimensions, it follows that tensor sum is defined for square matrices only. The algebraic definition of the tensor sum $\mathcal{Q} = \mathcal{Q}^{(1)} \oplus \mathcal{Q}^{(2)}$ are defined as:

$$\mathcal{Q}_{[ik][jl]} = q_{ij}^{(1)}\delta_{kl} + q_{kl}^{(2)}\delta_{ij},$$

where δ_{ij} is the element of the row i and the column j of an identity matrix, obviously defined as:

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

B.3 Classical Kronecker properties

Some important properties of the classical tensor product and sum operations are [2, 26]:

- Associativity:
  $\mathcal{Q}^{(1)} \otimes (\mathcal{Q}^{(2)} \otimes \mathcal{Q}^{(3)}) = (\mathcal{Q}^{(1)} \otimes \mathcal{Q}^{(2)}) \otimes \mathcal{Q}^{(3)}$ and
  $\mathcal{Q}^{(1)} \oplus (\mathcal{Q}^{(2)} \oplus \mathcal{Q}^{(3)}) = (\mathcal{Q}^{(1)} \oplus \mathcal{Q}^{(2)}) \oplus \mathcal{Q}^{(3)}$

- Distributivity over (ordinary matrix) addition:
  $(\mathcal{Q}^{(1)} + \mathcal{Q}^{(2)}) \otimes (\mathcal{Q}^{(3)} + \mathcal{Q}^{(4)}) =$
  $(\mathcal{Q}^{(1)} \otimes \mathcal{Q}^{(3)}) + (\mathcal{Q}^{(2)} \otimes \mathcal{Q}^{(3)}) + (\mathcal{Q}^{(1)} \otimes \mathcal{Q}^{(4)}) + (\mathcal{Q}^{(2)} \otimes \mathcal{Q}^{(4)})$

- Compatibility with (ordinary matrix) multiplication:
  $(\mathcal{Q}^{(1)} \times \mathcal{Q}^{(2)}) \otimes (\mathcal{Q}^{(3)} \times \mathcal{Q}^{(4)}) = (\mathcal{Q}^{(1)} \otimes \mathcal{Q}^{(3)}) \times (\mathcal{Q}^{(2)} \otimes \mathcal{Q}^{(4)})$

- Compatibility over multiplication:
  $\mathcal{Q}^{(1)} \otimes \mathcal{Q}^{(2)} = (\mathcal{Q}^{(1)} \otimes I_{n_{\mathcal{Q}}^{(2)}}) \times (I_{n_{\mathcal{Q}}^{(1)}} \otimes \mathcal{Q}^{(2)})$

- Commutativity of normal factors*:
  $(\mathcal{Q}^{(1)} \otimes I_{n_{\mathcal{Q}^{(2)}}}) \times (I_{n_{\mathcal{Q}^{(1)}}} \otimes \mathcal{Q}^{(2)}) = (I_{n_{\mathcal{Q}^{(1)}}} \otimes \mathcal{Q}^{(2)}) \times (\mathcal{Q}^{(1)} \otimes I_{n_{\mathcal{Q}^{(2)}}})$

Due to the Associativity property, the normal factor definition may be generalized to a tensor product of a suite of matrices, where all matrices but one are identities. This very useful normal factor definition can be applied to express more general forms of:

- Tensor sum definition as the sum of normal factors for all matrices:
  $$\mathcal{Q}^{(1)} \oplus \mathcal{Q}^{(2)} \oplus \mathcal{Q}^{(3)} = \begin{pmatrix} \mathcal{Q}^{(1)} \otimes I_{n_{\mathcal{Q}^{(2)}}} \otimes I_{n_{\mathcal{Q}^{(3)}}} \end{pmatrix} + \begin{pmatrix} I_{n_{\mathcal{Q}^{(1)}}} \otimes \mathcal{Q}^{(2)} \otimes I_{n_{\mathcal{Q}^{(3)}}} \end{pmatrix} + \begin{pmatrix} I_{n_{\mathcal{Q}^{(1)}}} \otimes I_{n_{\mathcal{Q}^{(2)}}} \otimes \mathcal{Q}^{(3)} \end{pmatrix}$$

*Although this property could be inferred from the *Compatibility with (ordinary matrix) multiplication*, it was defined by Fernandes, Plateau and Stewart [37].

- Compatibility over multiplication property as the product of normal factors for all matrices:

$$\mathcal{Q}^{(1)} \otimes \mathcal{Q}^{(2)} \otimes \mathcal{Q}^{(3)} = \begin{pmatrix} \mathcal{Q}^{(1)} \otimes I_{n_{\mathcal{Q}^{(2)}}} \otimes I_{n_{\mathcal{Q}^{(3)}}} \end{pmatrix} \times \begin{pmatrix} I_{n_{\mathcal{Q}^{(1)}}} \otimes \mathcal{Q}^{(2)} \otimes I_{n_{\mathcal{Q}^{(3)}}} \end{pmatrix} \times \begin{pmatrix} I_{n_{\mathcal{Q}^{(1)}}} \otimes I_{n_{\mathcal{Q}^{(2)}}} \otimes \mathcal{Q}^{(3)} \end{pmatrix}$$

The classical tensor algebra principles are applied since the first definitions of Stochastic Automata Networks (SAN). A SAN model is described as a sum of tensor products, following an algebraic formula called *descriptor* [55].

Appendix C

Notation

C.1 Stochastic Automata Networks

C.1.1 Basic Concepts and Definitions (Section 2.1)

✍ **Let be**

$\mathcal{A}^{(k)}$	the k^{th} stochastic automata in a network of K automata;
$s^{(i)}$	the i^{th} local state in an automaton;
$\delta^{(k)}$	the set of local states in an automaton;
n_k	number of local states in an automaton, *i.e.* the cardinality of $\delta^{(k)}$;
e_p	an event of ξ;
λ_p	the rate associated to the occurrence of an event e_p;
ξ	a set of P events, *i.e.* all events in the network of automata;
$\mathcal{X}$	the model state space; for structured models we denote it the product state space, whose the cardinality is given by $\prod_{k=1}^{K} n_k$;
$\mathcal{X}^R$	the model reachable state space, also denoted by $\mathcal{X}_{\tilde{s}_0}^R$, where $\mathcal{X}^R \subseteq \mathcal{X}$;
$\tilde{s}$	a global state inside the state space $\mathcal{X}$, *i.e.* a composition of local states of automata where $\tilde{s} = \{s^{(1)}; \ldots; s^{(K)}\}$;
$\mathcal{F}^R$	the reachability function of a model;

C.1.2 Graphical Representation and Primitives (Section 2.2)

✍ **Let be**

α_i	a constant rate associated to the occurrence of an event $e_p \in \xi$;
π_i	a probability associated to the occurrence of an event $e_p \in \xi$;
f_i	a functional rate associated to the occurrence of an event $e_p \in \xi$;
$\mathcal{F}^{R*}$	a partial reachability function of a model;

C.1.3 Structural Representation (Section 2.3.1)

✍ **Let be**

$\mathcal{Q}^{(k)}$	the k^{th} matrix in a tensor product of K matrices;
$\mathcal{Q}_j^{(k)}$	the k^{th} matrix in the tensor product j of K matrices;
$\mathcal{Q}_l^{(k)}$	the k^{th} matrix containing the local transitions rates and the diagonal adjustment in a tensor sum l of K matrices;
$\mathcal{Q}_{{e_p}^+}^{(k)}$	the k^{th} matrix containing the synchronizing event rate of e_p in a tensor product ${e_p}^+$ of K matrices;
$\mathcal{Q}_{{e_p}^-}^{(k)}$	the k^{th} matrix containing the diagonal adjustment for the synchronizing event e_p in a tensor product ${e_p}^-$ of K matrices;
$I_{\mathcal{Q}^{(k)}}$	an identity matrix of order n_k, *i.e.* with the same dimension of the matrix $\mathcal{Q}^{(k)}$;
e^+	a positive tensor product term $e^+ = \bigotimes_{k=1}^{K} \mathcal{Q}_{e_p^+}^{(k)}$;
e^-	a negative (diagonal adjustment) tensor product term $e^- = \bigotimes_{k=1}^{K} \mathcal{Q}_{e_p^-}^{(k)}$;
E	the number of synchronizing events in the model;

C.1.4 Structural Representation (Section 2.3.2)

✍ **Let be**

$\tilde{s}$ a global state inside $\mathcal{X}^R$;

$\tilde{r}$ a global state inside $\mathcal{X}^R$;

Φ a transition function associated to events changing global states;

$\Phi(\tilde{s}, e_p) = \tilde{r}$ the function Φ operation with a global state $\tilde{s}$ and an event e_p as input, and a global state $\tilde{r}$ as output;

$\tilde{s}_0$ initial global state of a trajectory generated by Φ successive applications considering a sequence of events $e = \{e_p\}_{p \in \mathcal{N}}$;

$\varpi(e_p)$ function to obtain the automata related to an event e_p;

α_p rate of an event $e_p \in \xi$;

C.2 Kronecker-based Descriptor Solution

C.2.1 Vector-Descriptor Product (Section 3.1)

✍ **Let be**

υ a probability vector of dimension given by the cardinality of $\mathcal{X}$;

$\mathcal{Q}$ a *descriptor* with a Kronecker representation;

$\bigotimes_{k=1}^{K} \mathcal{Q}_j^{(k)}$ a tensor product term composed of matrices $\mathcal{Q}_j^{(k)}$;

C.2.2 *Sparse* solution techniques (Section 3.1.1)

✍ **Let be**

$\mathcal{T}$ a tensor product term of K matrices $\mathcal{Q}^{(k)}$. It can be also seen as a huge matrix of dimension $\prod_{k=1}^{K} n_k$;

n_k dimension of the matrix $\mathcal{Q}^{(k)}$;

nz_k number of nonzero elements of the matrix $\mathcal{Q}^{(k)}$;

$\theta_{(1...K)}$ the set of all possible combinations of nonzero elements of the K matrices in a tensor product term;

a the scalar element generated through the combination of nonzero elements on a tensor product;

$nright_k$ the size of the state space corresponding to all matrices after the k^{th} matrix of the tensor product (special case $nright_K = 1$);

$nleft_k$ the size of the state space corresponding to all matrices before the k^{th} matrix of the tensor product(special case $nleft_1 = 1$);

π a probability vector of dimension given by the cardinality of $\mathcal{X}$. In the numerical methods is used as the stationary probability vector;

C.2.3 The memory-efficient *Shuffle* algorithm (Section 3.1.2)

✍ **Let be**

$\mathcal{T}$ a tensor product term of K matrices $\mathcal{Q}^{(k)}$. It can be also the tensor product term decomposed in normal factors (see Section B.1);

I_{nright_k} an identity matrix with dimension related to the size of the state space corresponding to all matrices at right of a given matrix $\mathcal{Q}^{(k)}$ in the tensor product;

I_{nleft_k} an identity matrix with dimension related to the size of the state space corresponding to all matrices at left of a given matrix $\mathcal{Q}^{(k)}$ in the tensor product;

z_{in} an auxiliary probability vector;

z_{out} an auxiliary probability vector;

υ a probability vector of dimension given by the cardinality of $\mathcal{X}$;

π a probability vector of dimension given by the cardinality of $\mathcal{X}$. In the numerical methods is used as the stationary probability vector;

C.2.4 The Hybrid *Split* Algorithm (Section 3.2)

✍ **Let be**

i_k the correspondent line in the matrix k;

j_k the correspondent column in the matrix k;

$\theta_{(1\ldots K)}$ the set of all possible combinations of nonzero elements of the K matrices in a tensor product term;

$\mathcal{T}$ a tensor product term of K matrices $\mathcal{Q}^{(k)}$. It can be also the tensor product term decomposed into an ordinary sum of matrices composed by one single nonzero element inside $\theta_{(1\ldots K)}$;

$\hat{q}^{(k)}_{(i,j)}$ a matrix of order n_k in which the element in row i and column j is $q^{(k)}_{i,j}$;

$\hat{q}_{(i_1,\ldots,i_{K-1},j_1,\ldots,j_K)}$ the matrix of order $\prod_{i=1}^{K} n_i$ composed by only one nonzero element, which is in the position $i_1,\ldots,i_K,j_1,\ldots,j_K$;

σ a *cut-parameter* of a given tensor product term. It is a division point for splitting a tensor product term in two separated parts ($\sigma = 1\ldots K$);

nz_i number of nonzero elements of the matrix $\mathcal{Q}^{(i)}$;

$\prod_{i=1}^{\sigma} nz_i$ total number of AUNF for a given *cut-parameter* σ;

a a scalar element generated through the combination of nonzero elements of matrices in a tensor product term. It is the scalar inside an additive unitary normal factor (AUNF);

$nright_\sigma$ the size of the state space corresponding to all matrices after σ in a tensor product term;

$nleft_\sigma$ the size of the state space corresponding to all matrices before σ in a tensor product term;

υ_{in} an auxiliary probability vector of dimension given by $nright_\sigma$;

υ_i an auxiliary probability vector;

υ a probability vector of dimension given by the cardinality of $\mathcal{X}$;

π a probability vector of dimension given by the cardinality of $\mathcal{X}$. In the numerical methods is used as the stationary probability vector;

C.2.5 Practical contributions of *Split* (Section 3.2.2)

✍ **Let be**

$\mathcal{T}$ a tensor product term which has $\sigma = 0 \ldots K$ as possibles division points;

ν number of samples or execution times collected from a given tensor product term, for each possible *cut-parameter* σ;

t_σ execution time t related to the *cut-parameter* σ of a tensor product term;

$\sigma_\mathcal{T}$ the assigned *cut-parameter* σ for a tensor product term $\mathcal{T}$;

C.3 Event-based Descriptor Solution

C.3.1 Forward Simulation (Section 4.1)

✍ **Let be**

$\tilde{s}$ a state in a system;

$\tilde{s}_n$ the n^{th} observed state of the system;

$\mathcal{X}^R$ the set of states considered in the system;

$\Phi(\tilde{s}, e_i) = \tilde{r}$ the function Φ operation with a state $\tilde{s}$ and an event e_i as input, and a state $\tilde{r}$ as output;

$\tilde{s}_0$ an initial state of a trajectory generated by Φ successive applications considering a sequence of events $e = \{e_i\}_{i \in \mathcal{N}}$;

τ^* the *warm-up* period, *transient* period or *burn-in time*;

π the stationary distribution;

ν number of samples or states generated;

C_s the simulation complexity cost;

c_Φ the complexity cost related to the function Φ;

C.3.2 Backward Coupling Simulation (Section 4.2)

✍ **Let be**

$\tilde{s}$	a state in a system;
$\mathcal{X}^R$	the set of states considered in the system;
τ	the coupling time;
$\mathbb{E}\tau$	the expected coupling time;
$-t$	a given time in backward steps;
C_s	the simulation complexity cost;
c_Φ	the complexity cost related to the function Φ;

C.3.3 SAN perfect sampling (Section 4.2.1)

✍ **Let be**

$\mathcal{X}^R$	the set of reachable states in the model;
$\tilde{s}$	a global state in a model;
e	an event generated;
$\Phi(\tilde{s}, e)$	the transition function Φ operating with a state $\tilde{s}$ and an event e;
$\phi(s^{(k)}, e)$	a transition function application in the local state $s^{(k)}$ of the k^{th} automaton operating an event e;
ω	a state vector to update the trajectories after Φ applications, initially filled with the states in $\mathcal{X}^R$;
$\tilde{\omega}$	a backup of the state vector ω;

C.3.4 Monotone Backward Coupling Simulation (Section 4.3 and 4.3.1)

✍ **Let be**

$\tilde{s}_{max}$	a maximal global state;
$\tilde{s}_{min}$	a minimal global state;
$\mathcal{X}^M$	the set of extremal states of the model, *i.e.* a set composed by maximal and minimal states in a partially ordered $\mathcal{X}$;
C_s	the simulation complexity cost;
c_Φ	the complexity cost related to the function Φ;
$\mathbb{E}\tau$	the expected coupling time;
E	an array that stores a backward sequence of events;

C.3.5 Extremal global states extraction (Section 4.3.2)

✍ **Let be**

M	a list of accessed states in the $\mathcal{X}^R$, initially storing the state $\tilde{s}_{min}$;
$M[i]$	the i^{th} position in the list M of accessed states;
nState	a new state generated;
cState	the current state;
e_p	an event in ξ to be fired over the current observed state cState;
Φ(cState ,e_p)	the transition function Φ operating with the current state cState and an event e_p;