PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
FACULTY OF INFORMATICS
COMPUTER SCIENCE UNDERGRADUATE PROGRAM

PEPS2015 - STOCHASTIC AUTOMATA NETWORKS SOFTWARE TOOL

LUIS GUSTAVO RAMOS ZANI

Final Undergraduate Work II submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fullfillment of the requirements for the degree of Bachelor in Computer Science.

Advisor: Prof. Dr. Paulo Lemelle Fernandes

Porto Alegre
2015

This work is dedicated to my parents and grandparents.

"There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies."
(C.A.R. Hoare)

ACKNOWLEDGMENTS

First of all, I am truly grateful to God for guiding me along the way and allow me to get to this point.

Among all people that have somehow supported me along these five years of undergraduate course, I would like to thank the ones that play an important role in my life, specially:

My parents, Mauro and Neiva, for all love and care they have always had for me, for supporting and encouraging me at all times. Everything that I've accomplished so far is because of you. I love you both.

My girlfriend, for all the pacience, support, and friendship in all moments. I know that many of them were not easy, but you were always there with me. Your love and care have always been very important for me.

My friend, Ramon Fernandes, for the imense help in several moments along the course. Thank you for all the very productive study hours and for being there whenever I needed a friend to talk to or a partner to discuss any kind of problem with.

My advisor, Paulo Fernandes, for the support. Your presence was important in some crucial moments along the way. I appologize for any incovenient I may have caused to you.

CNPq, for sponsoring my research project that has been developed along with this work.

The board of examinators, Prof. Afonso Sales (PUCRS) and Prof. Rafael Chanin (PUCRS), for all the constructive criticism that contributed to improve the quality of this work.

PEPS2015 - STOCHASTIC AUTOMATA NETWORKS SOFTWARE TOOL

ABSTRACT

This work presents the new version of PEPS software tool, designed for solving models expressed using Stochastic Automata Networks (SAN). The SAN formalism is historically the first Markovian structured formalism employing Tensor Algebra. PEPS tool was first launched in the late 80's and it's still a live project, therefore a short timeline of its previous versions, as well as the new features included in version 2015, are presented, such as an efficient and powerful just-in-time function evaluation.

Keywords: Analytical Modelling, Markovian Formalisms, PEPS, Stochastic Automata Networks, Systems Performance Evaluation.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

PUCRS – Pontifícia Universidade Católica do Rio Grande do Sul

FUW – Final Undergraduate Work II

SAN – Stochastic Automata Networks

PEPS – Performance Evaluation of Parallel Systems

MDD – Multi-Valued Decision Diagrams

CTMC – Continuous Time Markov Chain

DTMC – Discrete Time Markov Chain

CPU – Central Processing Unit

RSS – Reachable State Space

AUNF – Additive Unitary Normal Factor

GMRES – Generalized Minimal Residual Method

PSS – Product State Space

FAS – First Available Server

JIT – Just-In-Time

CONTENTS

1. INTRODUCTION

1.1 Analytical Methods

The analytical methods approach for systems performance evaluation consists in modeling a real system through its mathematical relations among the components. Using these mathematical relations, we can describe the system as a set of possible states and transitions between them with an associated behavior.

Among several existent methods, Markov Chains [1] are widely used to achieve stationary solutions for the models due to its large representation capabilities and simplicity on visualization and comprehension. However, this simplicity is also one of the biggest issues for using Markov Chains due to the well-known problem of state space explosion when modeling very large and complex models [33]. As more components are added to the model, the number of states in the corresponding Markov Chain increases dramatically, making harder to represent some models. Additionaly, the structure used to store the model is a square matrix of $n^t h$ order, where n is the number of states in the system. In case the matrix contains many zero elements, a sparse matrix could be used to store the model in order to reduce storage space. Moreover, not only there is the storage issue, but to solve the model, it's necessary to resolve the linear system of equations represented by the matrix, which also makes it difficult to obtain numerical results [31].

In order to work around the limitations of Markov Chains, another efficient approach that has been used by the cientific community and systems designers is the Stochastic Petri Nets [27]. This is a compact and efficient formalism to describe complex systems that contain synchronization and parallelism. It's said to be an efficient method because the solution of the problem is given by the product of its terms, which means it's not necessary to calculate the probability of each and every possible state to obtain the performance indexes of the model. However, it has been noticed that the actual impact of this formalism is on improving the modeling process, without adding significant improvements to the problem of achieving stationary solutions efficiently.

In that sense, Stochastic Authomata Networks (SAN) formalism proposes a compact way to describe complex systems and also optimizes the achievement of stationary solutions. SAN is a structured stochastic Markovian formalism first proposed by Plateau [30]. It allows us to represent an entire system by splitting it into a set of subsystems, each with an independent behavior itself (*local events), and also eventual interdependencies among each other (*synchronizing events and functional rates). In that sense, the framework proposed by Plateau estabilishes a structured and modular way to describe discrete and continuous-time Markovian models, but also emcompasses parallelism (when subsystems do not interact with each other) and synchronization (when subsystems interact with each other). SAN formalism is equivalent to Markov Chains in regards to scope of representation, but unlike them, SAN provides a model structure based on generalized tensor algebra [20], which makes it easier to achieve the stationary solution.

1.2 PEPS

PEPS project started in the late 80's and its goal was to develop a software package capable of computing numerical solutions for the SAN formalism. The first version of the tool was presented in [29] and featured a basic vector-matrix multiplication, where the matrix columns were generated one by one in each iteration. By using tensor algebra, only the tensor formula of the Kronecker descriptor needed to be stored, and the full matrix was never generated. Later on, another three official versions of the software tool were published in the cientific community.

Among several areas to which PEPS tool may be applied, it is convenient to cite the areas of computing and communication performance modeling, parallel and distributed systems, and finite capacity queueing networks [25]. PEPS differs from other tools supporting extended automata, such as UPPAAL [4] and KRONOS [2], in regards to scope, as the mentioned tools are more focused on modeling, validation and verification of real-time systems[1].

The core contribution of this project, in regards to PEPS latest version, is essentially a powerful and optimized function evaluation. This feature, as well as the techniques and algorithms implemented in the tool are shown in Figure 1.1, namely: a compact and efficient reachable state space (RSS) storage through the application of Multi-Valued Decision Diagrams (MDD), Shuffle and Split algorithms, and Tensor Algebra.

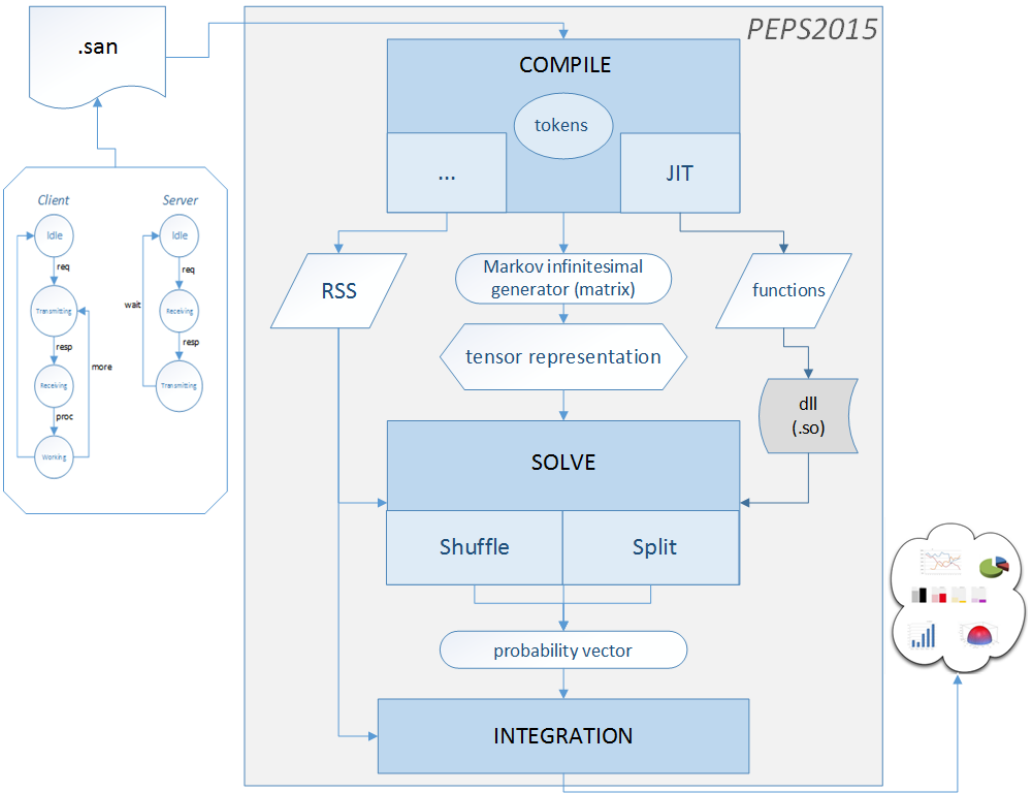

Figure 1.1 – Overview of PEPS 2015 modules and data

[1]Real-Time systems use networks of timed automata to model tasks that must be performed within strict time deadlines.

1.3 General Objective

The objective of this work is to explore the capabilities of the last published version of the software tool, aiming to identify opportunities of improvement both in the modelling process and system performance. For that, a detailed study of the current version was conducted along the year.

The general objective of the project is to apply scientific techniques to evaluate the capabilities of the developed software tool. The evaluations were done by means of the development of scientific articles. All of the proposed activities related to PEPS developed in this project will be available for the academic community as the newest version of the software tool, PEPS2015.

1.4 Specific Objectives

In a nutshell, the specific objectives that were achieved throughout this project are:

- Study the current implementation of PEPS software tool, version 2007.
- Study the algorithms and techniques used in PEPS2007.
- Study evaluation criteria for the performance of scientific software tools.
- Study other existent techniques, not currently implemented in PEPS2007, to determine whether they are relevant for the new version.
- Propose new features for a brand new version of PEPS tool.
- Define an implementation plan for the new features of PEPS, version 2015.
- Implement the proposed new features.
- Validate the implementation of the new features.
- Evaluate obtained results.

1.5 Document Structure

This document is organized as follows: Chapter I briefly contextualizes the reader on the subject, points out the objectives, and introduces the idea proposed in this project. Chapter II presents the motivation of this work, *i.e.* the description of the problem we intend to solve. Chapter III covers the theoretical foundation on which all the concepts involved in this work are based and presents a timeline of the related work that has been published so far. Chapter IV presents a detailed

explanation of the proposed solution for the problem described in Chapter II. Chapter V contains the analysis criteria defined to evaluate and validate the implemented features, as well as some testing scenarios and the final results. Finally, Chapters VI and VII expose our final conclusions and point out some ideas about the future work that can be done to improve the capabilities of the software tool.

2. PROBLEM DESCRIPTION

The last published version of PEPS, *i.e.* PEPS2007, is divided into two modules: one is responsible for the entire *compilation* process, and the other one takes care of the *execution* process of solving the model using numerical methods to try to achieve the stationary solution. The focus here will be in the *compilation* part of the tool.

The program takes a specific file containing the complete description of the SAN model as an input (which will be explained into details later in this document) and processes each section of that file during compilation, populating its internal data structures in order to store the model. The sections in the file contain the specifications of all components of the model, such as automata, states, transitions, rates, etc.

After a deep analysis of the whole compilation process step by step, we noticed that when the input model contains functional rates, the program uses a Stack-based data structure to store the functions internally. That creates a limitation for the tool in regards to the size of the models it can handle because it becomes dependent on the size defined for the stack. In other words, it means that if a model has many functional rates, which is quite common, the stack capacity might overflow before all functions get stored, making the tool unable to handle that model. Furthermore, the program also uses some temporary data structures during compilation to store information about the functions present in the model, which increases the memory usage and could eventually lead to an overflow as those data structures grow in size, although it's less likely to happen due to available memory resources we have nowadays.

Once all functions are finally stored, the program "translates" each of them into actual C-coded functions and creates a just-in-time (.jit) file containing every translated function in C language. The progam then calls GCC to compile the generated C code and produce a shared object that can be linked with other objects to form the executable. The shared object is used by the *execution* module in runtime to evaluate the functions when trying to solve the model.

A simplified scheme of the structures involved on storing the functions of the model during compilation can be seen in Figure 2.1.

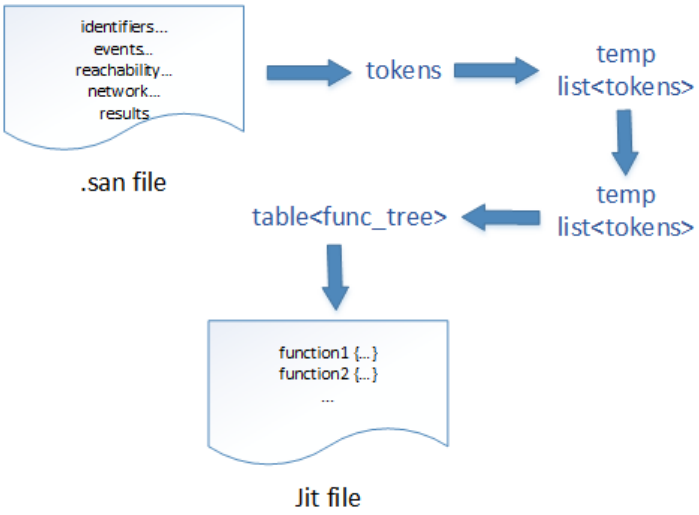

Figure 2.1 – Temporary data structures

With that in mind, we intended to propose a solution to work around the use of the stack when processing the functions present in the model and also try to eliminate the need of using the intermediate temporary data structures used for extracting data from the input file and move it to the permanent data structures. This way, the tool would supposedly use less memory in compile time and would be able to handle a much wider variety of models that were not supported in its previous versions.

3. RELATED WORK AND THEORETICAL FOUNDATION

3.1 Markovian Models

Unlike deterministic processes, which can only evolve in one way (like the solution of a differential equation, for example), stochastic processes consist of a set of random variables, representing the evolution of a model with such random variables over time. A stochastic process contains a certain level of indeterminacy in a way that even if the initial state is known, there are several different directions in which the process may evolve. Some common examples of stochastic time series include stock market, exchange rate variations, and audio/video signals.

Markovian models are stochastic models used for modeling systems that possess randomly changing behavior with an exponential distribution, *i.e.* events occur continuously and independently at a constant average rate. The future states depend exclusively on the present state of the model and not on the sequence of events that preceded it, there is no memory. This memoryless property is known as the *Markov property*. Generally, it enables us to solve problems that would otherwise be solvable only in theory, given a large but finite time, but which pratice would take too long for the solution to be useful, also known as *intractable* problems.

The theory of Markov processes can be divided into subtheories that depend on whether time is discrete or continuous or whether the set of possible states is finite or countably infinite. In this case, we speak about *Markov chains*. If the set of possible states has cardinality of the continuum, we speak of *continuous Markov processes*.

A Markov chain can have transitions between states occuring at discrete moments in time, and in this case is called a *Discrete Time Markov Chain (DTMC)*; or it can have transitions occuring at any moment, with a continuous set of times, and so is called a *Continuous Time Markov Chain (CTMC)*. Since Markovian Chains are not the focus in this work, if the reader is interested in more details about DTMC and CTMC, it can be found in [11].

3.2 SAN

Stochastic Automata Networks is a Markovian formalism designed for modeling systems with large state spaces. It consists in modeling a system as various subsystems, with eventual interactions among the modules. This modularization defined by SAN enables complex systems to be stored and solved efficiently, for it avoids the explosion on the number of states that happens when using Markov Chains, to which SAN is equivalent in regards to representation.

Each subsystem is defined as an individual stochastic automaton and transitions between the states of this automaton. The transitions between the states of each automaton are modeled

by a stochastic process in the continuous or discrete time scale defined by exponential or geometric distributions, respectively. In this work, we will be focusing on continuous time models.

It's important to mention that every SAN can be represented by a single stochastic automaton that contains all possible states of the system. This automaton corresponds to the equivalent Markov Chain for that SAN model.

3.2.1 Stochastic Automata

A stochastic automaton is mathematical model of a system that contains descrete inputs and outputs. The system can be in any state of a finite set of states defined for that system. An internal state of the system summarizes the information about previous inputs and indicates what is necessary to determine the behavior of the system for the following inputs. In that sense, a stochastic automaton can be defined as a finite set of states and a finite set of transitions between those states. It is called stochastic because the time is treated as a random variable that follows an exponential distribution in the continuous time scale.

A state in the SAN model is called a *global state* of the system and consists of the current individual state of each automaton in the model [30]. A change in the global state of the system is given by the change of a local state in any of the compounding automata in the model. A change in the local state of an automaton is given by a transition. Transitions are elements that indicate the possibility of changing from state to another, however each transition must have an associated *event* in order to be triggered. There can be any number of automata in a SAN model.

Figure 3.1 shows an example of a SAN model with two completely independent automata.

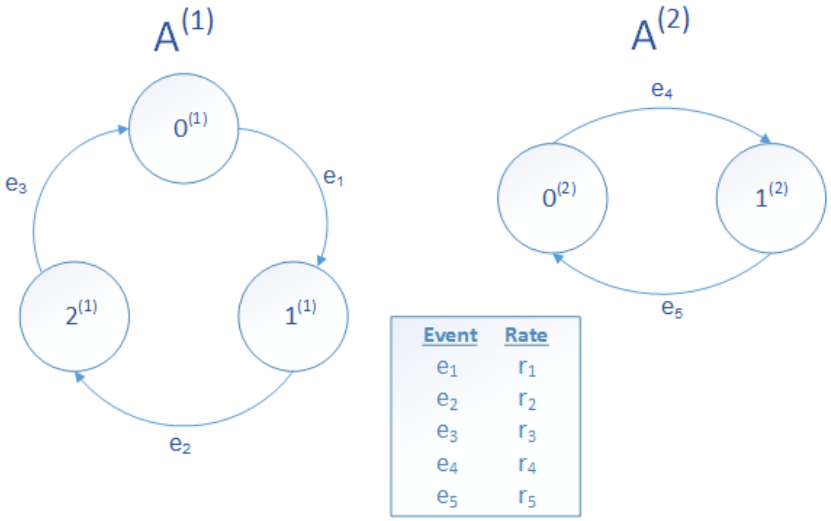

Figure 3.1 – Example of a SAN model with independent automata

In the example presented in Figure 3.1, the automaton $A^{(1)}$ contains three local states $0^{(1)}, 1^{(1)}$ and $2^{(1)}$, and the automaton $A^{(2)}$ contains two states $0^{(2)}$ and $1^{(2)}$. Three of the five

events in the model (e_1, e_2 and e_3) occur in $A^{(1)}$, while the other two (e_4 and e_5) occur in $A^{(2)}$. Considering the event rates shown in the table, Figure 3.2 depicts the equivalent CTMC to the SAN model presented in Figure 3.1. Notice that there are only *local events* in each automata in the model, therefore there is no interaction whatsoever between them.

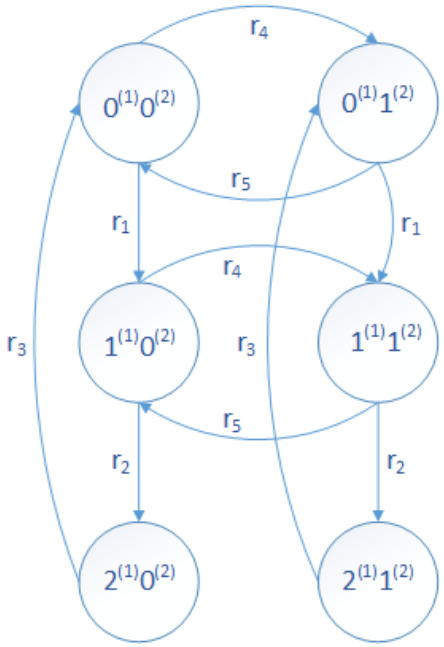

Figure 3.2 – Continuous Time Markov Chain equivalent to SAN model with 2 independent automata in Figure 3.1

3.2.2 Events

An event is the ocurrence of a transition that changes the *global state* of the model, either because of a change in a local state of an automaton (local events) or because of a synchronized change in two or more autoamta (synchronizing events). Each transition must have one or more associated events and it is triggered whenever any of those events occur.

Local events are used in SAN to alter the local state of a single automaton, without affecting any other automaton in the model. This type of event allow various automata to work in parallel, not interfering on each other's behavior, with no interactions between them. Some examples of local events can be seen in Figure 3.1, in which all events are this type.

Synchronizing events, on the other hand, are the ones used in SAN for altering the local state of two or more automata simultaneously, *i.e.* when a synchronizing event occurs in one automaton, it necessarily also occurs in all the other automata involved in that event. Therefore, the interaction between automata is given by this type of event, in the form of synchronization when triggering transitions.

An event is considered *local* if it only appears in the set of transitions of only one automaton and *synchronizing* in case the same event appears in the set of transitions of two or more automata when the model is designed.

Every event must have an associated firing rate, to which can be assigned either a constant or functional value. Functional rates can have different values, depending on the state of the other automata in the model.

3.2.3 Functional Rates

Apart from the synchronizing events, the other way automata can interact is through functional rates. By using functions to define transition rates it is possible to assign different values to the same event, according to the global state of the model.

Functional rates are expressed by functions that take into account the current states of the automata in the model, so the value can vary depending on the state that each automaton involved in the function is currently in.

In the example shown in Figure 3.3, there are also 2 automata and 5 events, similarly to the example in Figure 3.1, but in this case, the event e_5 was turned into a synchronizing event, once it appears on both automata and it has different probabilities associated to each transition in automaton $A^{(1)}$. Event e_5 has also changed, since it now has the function f_1 associated to the firing rate of the event.

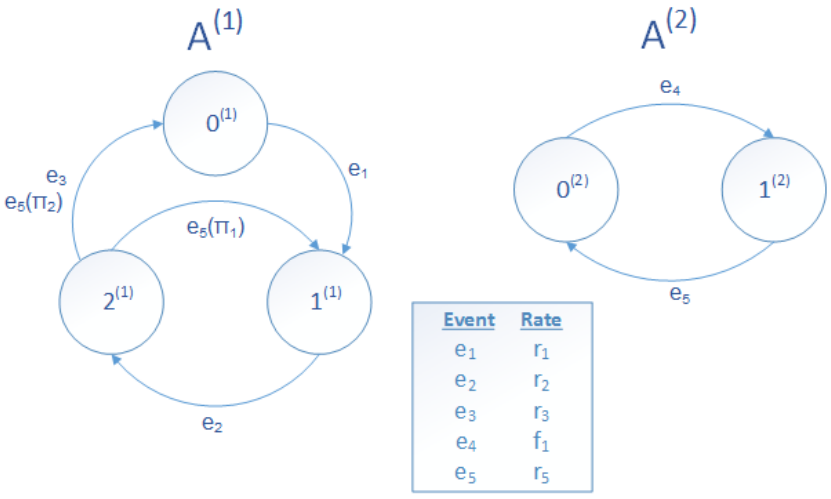

Figure 3.3 – Example of a SAN model with synchronizing events and functional rates

Function f_1 shown in Figure 3.3 is defined as:

$$f_1 = \begin{cases} \lambda_1 & \text{if automaton } A^{(1)} \text{ is in state } 0^{(1)}; \\ 0 & \text{if automaton } A^{(1)} \text{ is in state } 1^{(1)}; \\ \lambda_2 & \text{if automaton } A^{(1)} \text{ is in state } 2^{(1)}; \end{cases}$$

This means that the firing rate of the transition from state $0^{(2)}$ to $1^{(2)}$ is λ_1 in case automaton $A^{(1)}$ is in state $0^{(1)}$ or it is λ_2 in case automaton $A^{(1)}$ is in state $2^{(1)}$. If the state if automaton $A^{(1)}$ is $1^{(1)}$, then the transition will never occur.

The CTMC equivalent to the SAN model shown in Figure 3.3 can be seen in Figure 3.4.

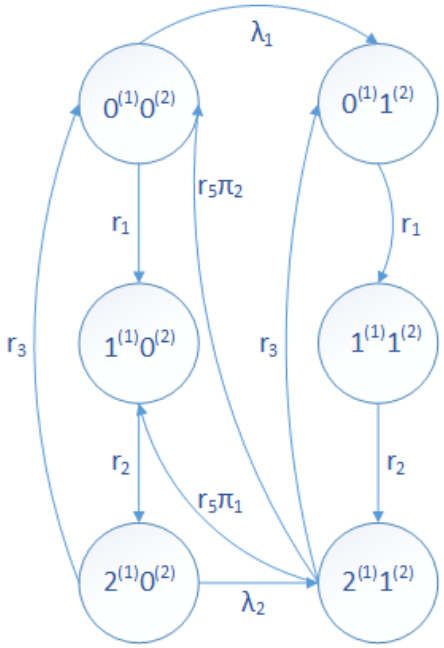

Figure 3.4 – Continuous Time Markov Chain equivalent to SAN model in Figure 3.3

3.2.4 Reachability Function

Another type of function that is used in SAN modeling is called the *Reachability Function*. This function is described using the same rules applied to the specification of functional rates, but it plays a completely different role in the model.

Since the representation of a model in SAN format is modularized and the global automaton that corresponds to the equivalent Markov Chain is given by the combination of all automata in the model, it is necessary to specify a function that defines which of the states of this global automaton representing the SAN can actually be reached.

The definition of which states can be reached in SAN is given by the reachability function. It is easier to understand the concept of reachability with an example of a resource sharing model,

in which there are N processes trying to access R resources. For any R smaller then N, the global state that represents all N processes acessing one resource would be an *unreachable state*, for it does not belong to the reality of the model it has to be eliminated from the model by the reachability function because the probability of the model to be in this state is equal to 0.0.

3.2.5 Integration Functions

The same way we define functional rates and the probability function, it is also necessary to define *Integration Functions* for a SAN model. These are the functions that evaluate the probability of the SAN model to be in a specific state.

This way, it is possible to build compound integration functions that evaluate the probability of the model being in a set of states, allowing us to obtain some performance and reliability indexes for the model.

The integration functions are evaluated based on the *probability vector*, that contains the probability of the model to be in each of the states in the vector.

An example of an integration function, considering the example shown in Figure 3.1, is given by g, which indicates the probability of automaton $A^{(2)}$ being in state $0^{(2)}$:

$$g = st(A^{(2)}) == 0^{(2)}$$

Roughly, all funtions are defined the same way in SAN. The only difference is how they are applied in the model. The formal definition of the integration functions for SAN models can be found in [9].

3.2.6 SAN Example

In order to clarify the presented concepts so far, this section presents an example of an arbitrary context that can be modelled with SAN formalism.

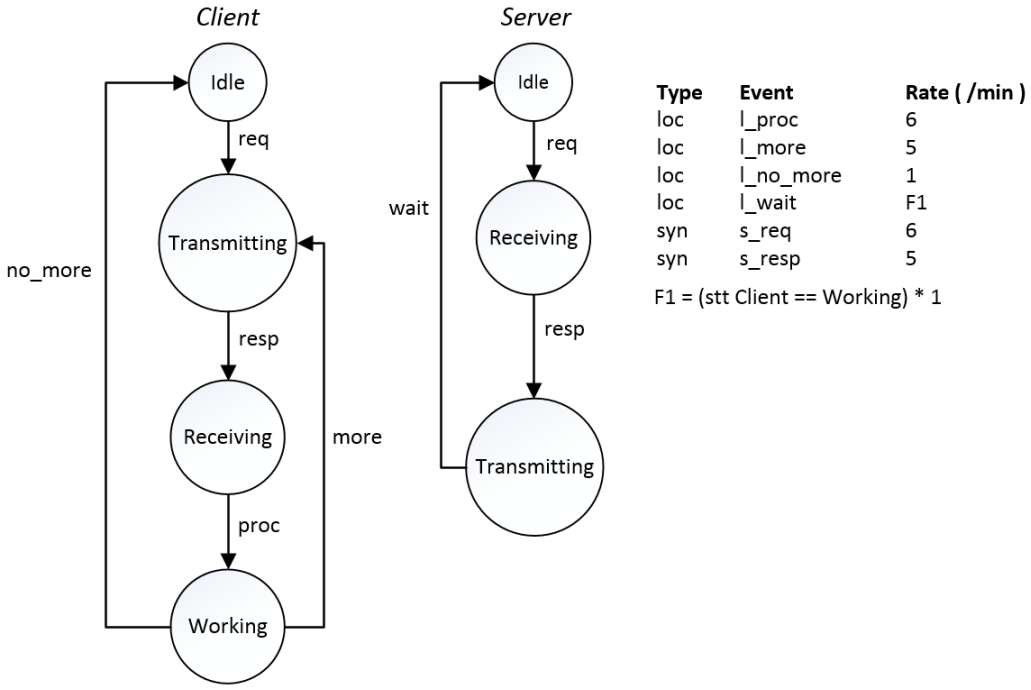

Figure 3.5 – Example of a SAN model

The SAN model in Fig. 3.5 describes a Client-Server system as a composition of two subsystems. The first one, named *Client*, contains 4 local states: *Idle*, *Transmitting*, *Receiving*, and *Working*; and the second one, called *Server*, has three local states: *Idle*, *Transmitting*, and *Receiving*, in order to represent the basic functioning of the respective components. Notice that the *local events* defined in this model only change the state of one automaton at a time, without affecting any other indeed. In this model, the local events are named *proc*, for "process", *more*, for "more data to be transmitted", *no_more*, for "nothing else to be transmitted", and *wait*, for "wait for requests". Also notice that the *synchronizing events* occur simultaneously in more than one automaton, making all of the affected automata change their states at the same time. In this case, the synchronizing events are named *req*, for "request", and *resp*, for "response". They change states from "Transmitting" to "Receiving" on the Client side, and vice-versa on the Server side with only one occurrence.

The *functional transition rates* determine that the average firing rate of the events is not a constant value [1]. In fact, there is a function (in this case, *F1*) that determines the possible values for this rate, depending on the state of the other automaton. In the example depicted in Fig. 3.5, the functional transition rate *F1* indicates that the Server only changes states from "Transmitting" to "Idle" when the Client is at the state "Working", meaning that while there is communication between the components, the server cannot go idle.

[1] Since it is a Markovian Formalism, SAN assumes that all rates represent the average of an exponential distribution. Hence, a constant rate stands for the average frequency of an exponentially distributed phenomenon [18].

3.3 PEPS 2000

One of the earliest versions of PEPS project was released in the year 2000. The main features implemented in that version are described hereafter:

3.3.1 Textual Input

PEPS software tool takes files with the extension *.san* as an input. These files contain the detailed description of every component of a SAN model that is to be numerically resolved by the tool. The specification of a SAN is basically composed of five sections: *identifiers, events, partial reachability, network*, and *results*. An example of a *.san* file that represents the model depicted in Fig. 3.5 is presented in Chapter 3.7 of this document.

- **Identifiers**: The first section of a *.san* file is where the average firing rates for all the events in the model are defined. It's important to note that each rate is required to have a unique name, *i.e.*, an identifier. As mentioned before, each firing rate can be assigned either a constant value or a function.

- **Events:** In this section, all events present in the model must be described. For each event, its type (local or synchronizing) and name are specified, as well as which identifier its firing rate corresponds to. Each event firing rate is associated with the identifier for that specific rate or to a function that represents a functional transition rate.

- **Partial Reachability:** This is the section where the reachability function is specified. It is basically a boolean function that returns 1 if the state is reachable and 0 in case it is not. The word *partial* indicates that the expression used for describing the set of reachable states encompasses only part of states, not all of them. The software tool is able to discover the remaining reachable states starting from the given set and generate the complete reachability function.

  The expression defined in this section can also be seen as the definition of the starting state of each automaton. Since not all global states are reachable, the combination of the starting states is specified to be surely reachable. In other words, it is guaranteed that at least this global state is known to be a reachable state in the model.

- **Network:** In this section of the .san file, the model itself must be described. Therefore, the name of the model is defined, along with the time scale being used (PEPS only supports *continuous* models). Then, all of the automata in the model are specified using a hierarchical structure. It includes names, states, and the transitions associated with their corresponding firing rates for each automaton. An example of how states and transitions are defined can be seen in Chapter 3.7.

- **Results:** In this section, the functions for obtaining the performance indexes intended for the model, *i.e.* integration functions, are defined. The results of these functions will indicate the probability of the model to be in that state or set of states defined by the function.

3.3.2 Shuffle Algorithm

The main increasing technique implemented in this version was a vector-descriptor multiplication method proposed by Fernandes [20] named *Shuffle* algorithm.

The Vector-Descriptor product is one of the most important operations to achieve both stationary and transient solutions for models described by Kronecker structured formalisms using iterative methods. The Shuffle algorithm was implemented to perform this kind of operation in the model. In this method each part of the probability vector is multiplied by a tensor product term, never generating any part of the full matrix. After all multiplications are done, we have a complete vector-matrix multiplication.

Essentially, it consists in exploiting the property of decomposing a tensor product into the ordinary product of tensor products in a normal form.

$$\begin{aligned} Q^{(1)} \otimes ... \otimes Q^{(N)} = (Q^{(1)} \otimes I_{n2} \otimes ... \otimes_{nN-1} \otimes I_{nN}) \\ (I_{n1} \otimes Q^{(2)} \otimes ... \otimes I_{nN-1} \otimes I_{nN}) \times \\ . \\ . \\ . \\ (I_{n1} \otimes I_{n2} \otimes ... \otimes Q^{(N-1)} \otimes I_{nN}) \times \\ (I_{n1} \otimes I_{n2} \otimes ... \otimes I_{nN-1} \otimes Q^{(N)} \end{aligned} \tag{3.1}$$

Rewriting the basic operation of the vector-descriptor product, *i.e.,* the required number of multiplications of vector v by a tensor product of N matrices, according to this property:

$$v \times \left[\prod_{i=1}^{N} I_{nleft_i} \otimes Q^{(i)} \otimes I_{nright_i} \right] \tag{3.2}$$

where *nleft* corresponds to the product of the order of all matrices before the i^{th} matrix of the tensor product term, *i.e.,* $\prod_{k=1}^{i-1} n_k$ (particular case: $nleft_1 = 1$) and $nright_i$ corresponds to the product of the order of all matrices after the i^{th} matrix of the tensor product term, *i.e.,* $\prod_{k=i+1}^{N} n_k$ (particular case: $nright_N = 1$) [21].

This way, In other words, the Shuffle algorithm consists in the successive multiplication of a vector by each normal factor. In fact, vector v is multiplied by the normal factor, then the resulting vector is multiplied by the normal factor again, and so on until the last factor. The multiplication

of a vector v by the i^{th} normal factor is equivalent to *shuffle* the elements of v in order to assemble $nleft_i \times nright_i$ vectors of size n_i and multiply them by matrix $Q^{(i)}$. So, assuming the matrix $Q^{(i)}$ is stored as a sparse matrix, the number of multiplications necessary to multiply a vector by the i^{th} normal factor is:

$$nleft_i \times nright_i \times nz_i \tag{3.3}$$

where nz_i is the number of nonzero elements of the i^{th} matrix of the tensor product term $Q^{(i)}$. Considering the number of multiplications to all normal factors of a tensor product term, the result is [14, 21]:

$$\prod_{i=1}^{N} n_i \times \sum_{i=1}^{N} \frac{nz_i}{n_I} \tag{3.4}$$

The reader may find extensive material in the literature [14] in regards to memory and CPU efficiency of the Shuffle algorithm.

3.4 PEPS 2003

Another version of the tool was released in the year 2003 with some improvements over PEPS2000 version:

3.4.1 Lex/Yacc Compiler

One of the new features that PEPS2003 brought was a new interface compiler. It became more intuitive and compact and implemented some basic replication features to describe identical automata and queueing structures.

The elements of the new compiler as a whole, such as the grammar and the algorithms used to parse the input were generated through the use of existing specialized tools. The lexical symbols of the grammar and the rules applied in the lexical analysis performed by the compiler, such as splitting the source file into tokens, were generated by Lex [3] tool. The rules applied on finding the hierarchical structure of the program, *i.e.* the syntactic analysis, were defined using Yacc [5] tool. Hence, the new compiler for the 2003 version was implemented based on the use of Lex and Yacc tools.

3.4.2 Integration Functions

The final steps of the Shuffle algorithm, implemented in the 2000 version, were responsible for calculating the probability of the model being in each global state and populate the probability vector. In that sense, one of the new implemented features for this version was the integration

functions. As mentioned previously, these functions sum the product of each row of the probability vector containing all the global states in order to obtain the results. Since not all global states will necessarily be reachable, some of these probabilities will be zero. This feature improved the quality of the analysis that could be performed on the model solved by extracting performance indexes and created room for more accurate conclusions.

3.5 PEPS 2007

PEPS2007 is the latest version of the software tool. It is currently available for the academic community [10] and it contains some of the newest features implemented in the tool, namely:

3.5.1 Modularized Software Structure

In this version, a new software structure was adopted in order to improve PEPS maintenance and development, since the project would keep getting bigger and more complex throughout the time. The new structure divided PEPS2007 into a set of individual modules. Each model implemented a part of the compiling/solution procedures. Basically, PEPS modules were assembled in three groups: *Interface, Data Structure*, and *Solution Methods*.

With this approach, it was possible to develop and test new methods independently of the software stable version. In other words, different versions could be developed in a locally and/or logically distributed way, without any particular care about eventual bug corrections and improvements in other parts of PEPS software.

3.5.2 New Compiler

The development team came up with some improvements for the compiler in the 2007 version. It was meant to facilitate the description of more complex models with replicated automata, or states inside an automaton. It kept the full backward compatibility with PEPS2003 and it was based on the previous version interface.

The two main differences were the ability to replicate not only identical structures, but also automata and states with different transitions, events, and rates (with the restriction that replicated automata must have the same cardinality, *i.e.* same number of states), and the actual implementation of lexical, syntactic and semantic analysis of the compiler. For this version, instead of using Lex/Yacc generators, the entire grammar, as well as syntax and sematic rules were reimplemented manually using C++ programming language.

3.5.3 Bounding Methods

Tensor algebra can be used in SAN to store large Markov Chains in a compact format. However, in some cases it is still hard to compute the model solution for really huge models and calculating the exact performance indexes becomes not feasible. The new feature in PEPS2007, *Bounding* algorithms [23], can be used to obtain upper or lower bounds of performance in Markov Chains. This method reduces the Markov Chain transition matrix to a smaller monotone matrix. It was implemented in this version in such a way that the software tool generates a monotone upper bounding matrix directly from the Markovian Descriptor that represents the SAN model.

3.5.4 Automata Aggregation

Automata aggregation in SAN is meant to reduce the number of automata in the model, bringing some numerical benefits to the solution. According to the method described by Benoit *et al.* [7], a group of fully identical automata, *i.e.* , automata with the exact same number of states, transitions and rates, may be aggregated into a single automaton where each state represents the number of grouped automata in each of the states.

Among the numerical benefits that were brought with this feature, it is convenient to note the theoretical and practical results, which were a quite significant decrease of total state space and reduced memory usage, respectively.

Alongside with the reduction of the total state space, additional benefits consequently come up, such as the elimination of some functional rates and unreachable states. It occurs because of the nature of the aggregation methods, which basically groups automata that are connected to the same state(s) into only one automaton.

There were essentially two aggregation methods for SAN formalism. The *algebraic* aggregation method uses properties of the generalized tensor algebra to reduce the number of automata and the *semantic* aggregation method is based on the relationship among replicated automata [7].

3.6 PEPS 2009

In 2009, some new features were proposed, aiming to improve the solving process in general with new techniques. However, these features have not been fully implemented in the software tool, therefore this version has never been published and it is **not** available for the scientific community.

The proposed features for this version, which are still under development, are described hereafter:

3.6.1 Reachable State Space Efficient Generation

A Multi-valued Decision Diagram (MDD) [24] is a compact structure that allows us to store and to manipulate large sets of structured information in a compact format. The information in a model can be structured by N components (or subsystems, *i.e.* , in our case, *automata*) where these components have some independent behavior and occasional interdependency.

Basically a MDD is a directed acyclic edge-labeled multi-graph, where there are some specific properties, such as: (i) nodes are organized into $N + 1$ *levels*, where N represents the number of subsystems; (ii) level N has only one single *non-terminal* node (known as the *root*), whereas levels $N - 1$ through 1 have one or more non-terminal nodes; (iii) level 0 has two *terminal* nodes: 0 and 1; (iv) a non-terminal node p at level l contains n_l arcs pointing to nodes at level $l - 1$, where n_l indicates the number of local states of l-th subsystem; (v) there are not *duplicate* nodes, *i.e.* , nodes at a same level are unique.

Fig. 3.6 illustrates a MDD which represents the state space $\mathcal{S}$, subset of a cross-product of a system splitted in four subsystems (*i.e.* , $N = 4$), where $\mathcal{S}^{(i)}$ represents the local state space of the *i-th* subsystem.

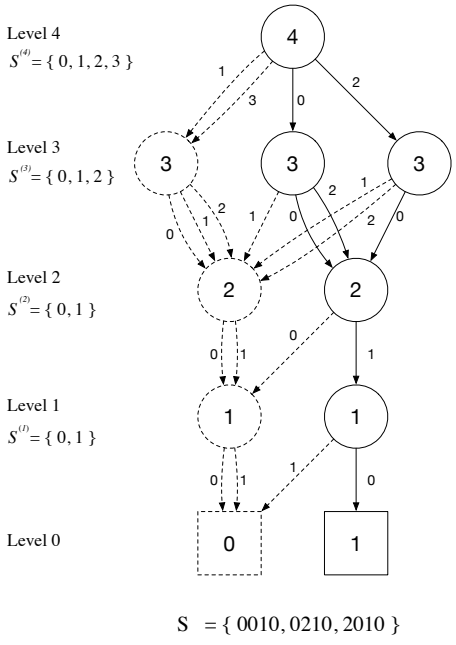

Figure 3.6 – A state space $\mathcal{S}$ represented by a MDD

In Fig. 3.6, *non-terminal* nodes are depicted by *circles* and *terminal* nodes are depicted by *squares*. A given state x of the system is composed by the combination of the local states of each subsystem, *i.e.* , $x = x^{(N)} ... x^{(1)}$ of $\mathcal{S}$ is element of a subset represented by a N-level MDD if and only if the path through the MDD, starting at the level-N node, following downward pointer $x^{(l)}$ at level l, reaches terminal node 1. Dashed arcs and nodes are paths which lead only to terminal node 0 and in the rest of the paper, for reasons of clarity, will be omitted.

Also, MDD can be used to represent a set S of integer tuples by storing the characteristic function f_S of the set. Therefore sets can be manipulated using operations over MDDs on their characteristic functions (*e.g.*, the operation union on sets corresponds to disjunction on MDDs).

Saturation-based state-space generation is a successful symbolic method based on MDDs applied to generating state spaces of structured models, *e.g.*, Stochastic Petri Nets (SPN) models [12, 13, 28].

Sales and Plateau [32] presented an extension of the saturation method described in [12], which allows the use of *functional transitions*, *i.e.* , the interaction between automata can be represented by a function. Functions can be associated to rates or routing probabilities on the events. In this case, the rate or the routing probabilities of an event can have different values in function to the state of other automata.

The main idea of the generation method is to compute the reachable space state (RSS) of the model by the successive firing of events from an initial state while the RSS is stored in a MDD. The method exploits the possibility of firing any event that affects a given MDD node and its descendant nodes, thus bringing the node to its *saturated* format. Besides, nodes are considered in a *bottom-up fashion* (*i.e.* , when a node is computed, all its descendant nodes are already in the *saturated* format). A node is considered as *saturated* if it encodes a set of states which are a fixed point in regards to the firing of any event at its level or at a lower level.

Fig. 3.7 shows a small SAN model with three automata ($N = 3$) and the corresponding RSS of this model re presented by a MDD, generated from initial state 000.

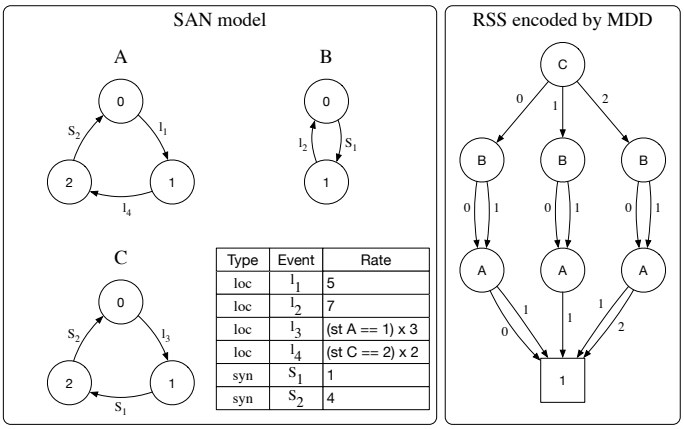

Figure 3.7 – RSS of the SAN model represented by a MDD

Functions are a powerful feature in the SAN formalism, since it allows us to represent very complex behaviors of a system in a very compact format. Moreover, the usage of MDDs to generate the RSS of a SAN model, which uses functions (having a small domain size), allows us to achieve the generation of large reachable state spaces with a small computational time, while keeping a low memory consumption.

3.6.2 Split Algorithm

SAN schemes that model real scenarios are naturally sparse: That's because the tensor sum structures generally make the local portion of the descriptor very sparse. Also, the synchronizing events make the descriptor quite sparse, since they are mostly used to describe exceptional behaviors.

The *Split* algorithm [14] allows each tensor product of matrices to be partitioned into two different groups: one with more sparse matrices and the other with more nonzero elements. This way, the Sparse algorithm [15] could be applied to the first group generating *Additive Unitary normal Factor* (AUNF). An AUNF is composed of three elements: a scalar value obtained by multiplying one nonzero element of each matrix in the Sparse-like part by each other; an input slice, which is a part of the vector v identified by the line row coordinates i of the nonzero elements multiplied; and an output slice, as a part of the vector v identified by the column coordinates j of the multiplied elements [14]. Each of the generated AUNF must then be multiplied (tensor product) by the second group of matrices using the Shuffle algorithm, as Slice handles only the first matrix in this case.

Considering this concept, the goal is to split the tensor terms into two sets of matrices and deal with them in different ways, almost individually. Therefore, the *Split* algorithm is considered a generalization for the encompassing all possibilities from a pure Sparse approach until the *Shuffle* algorithm, once it follows the idea of breaking the system into distinct parts to be dealt with.

In regards of efficiency, a deeply detailed material may be found in the literature [14], explaining the memory and CPU efficiency of the Split algorithm, as well as Sparse and Slice algorithms. Additionally, some optimizations on the Split algorithm were proposed by Czekster, Fernandes, and Webber in [15], in case the reader is interested.

3.7 File (.san) Example

In order to make the way PEPS tool works more explicit to the reader, this chapter presents an example of a *.san* file, along with its compilation and execution processes.

The code shown in Code 3.1 is a demonstrative example of the *.san* file that describes the SAN model depicted in Fig. 3.5, containing all of the components presented in Chapter 3.3.1 of this document.

```
identifiers

  //rates = per minute
  r_proc    = 6;
  r_more    = 5;
  r_no_more = 1;
  r_req     = 6;
  r_resp    = 5;

  F1 = (st Client == Working) * 1;
```

```
events

  loc l_proc    (r_proc);
  loc l_more    (r_more);
  loc l_no_more (r_no_more);
  loc l_wait    (F1);
  syn s_req     (r_req);
  syn s_resp    (r_resp);

// Both Client and Server start in idle.
partial reachability = ((st Client == Idle)
                    && (st Server == Idle));

network ClientServer (continuous)

  aut Client
    stt Idle         to (Transmitting)  s_req
    stt Transmitting to (Receiving)     s_resp
    stt Receiving    to (Working)       l_proc
    stt Working      to (Transmitting)  l_more
                     to (Idle)          l_no_more

  aut Server
    stt Idle         to (Receiving)    s_req
    stt Receiving    to (Transmitting) s_resp
    stt Transmitting to (Idle)         l_wait

results

  Client_requesting = (st Client == Transmitting);
  Client_receiving = (st Client == Receiving);
  Client_processing = (st Client == Working);
  Client_trans_Serv_rcv = ((st Client == Transmitting )
                       && (st Server == Receiving));
  Client_recv_Serv_trans = ((st Client == Receiving )
                        && (st Server == Transmitting));
  Client_idle = (st Client == Idle);
  Server_idle = (st Server == Idle);
```

Code 3.1 – Example of the SAN file for Client-Server model

For this example, the performance indexes that the user expects to know are specified in the results section, *i.e.* , the probability of the Client being requesting, receiving, and processing. Besides, the percentage of the time the Client is transmitting data and the Server is receiving data and vice-versa are also desired results. At last, the percentage of time that both Client and Server are idle.

3.8 Usage

An example of usage of the software tool is shown in this chapter. Both compilation and resolution of the *.san* file presented in the previous chapter can be seen in Fig. 3.8. For this

demonstration, the compilation process does not make use of any aggregation method [7] and the resolution process uses the Power Method [26] to solve the model[2].

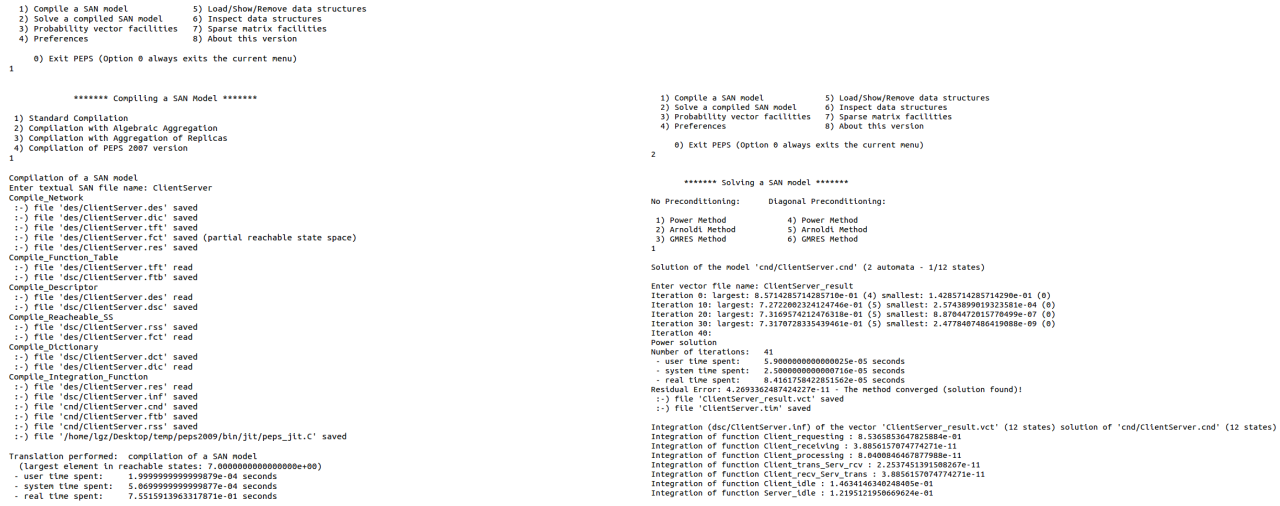

Figure 3.8 – Compilation and resolution of the Client-Server SAN model

According to the output of the system, shown in the right hand side of Fig. 3.8, the probability of both Client and Server to be idle is less than fifteen percent, which means that, for these parameters, the model presents satisfying results in regards to occupancy, for the Server is not under used nor overloaded.

3.9 Supported Models

In regards to the previous versions that have been developed along the years since PEPS project started, some criteria have been defined in order to evaluate the capabilities of the tool when running different types of stochastic models.

Since Markovian models can easily become large and complex, the project team defined as reasonable analysis criteria the total size of the model itself, *i.e.* , the number of states or *Product State Space* (PSS), and the *total memory used* to solve the model, considering how big the matrixes and internal structures grow, and finally the *total execution time* spent to solve the model.

[2]PEPS versions of 2003, 2007, and obviously 2015 also offer the possibility to perform stationary solutions using also Arnoldi and GMRES methods, and also transient solution using Uniformization methods [34].

Some of the types of models that have been proven to be supported by the tool are presented in Table 3.1. It presents the numerical results considering the analysis criteria mentioned above in order to show that the tool can handle the models within reasonable time and space limits for each **published** version of the tool.

Table 3.1 – PEPS numerical results

PEPS version	**Model**	**PSS**	**Memory Used**	**Time to Solve**
2000	General Resource Sharing [17]	1,084,576	590 Bytes	33 sec.
2003	FAS - First Server Available [8]	33,554,432	11.2 KB	5.47 min.
2007	Master-Slave Parallel Program [6, 14]	65,367,200	38.54 KB	8.92 min.

Based on the presented data, it is interesting to note that the number of states in the network seems to be the driving factor for the final solution time. For the latest version of the tool (PEPS2007), the execution time is fairly reasonable, although it bit longer, for it resolves a model with almost twice as many states as the previous version, using an acceptable amount of memory.

4. PROJECT DESCRIPTION

4.1 Architecture

For the new version of PEPS, the architecture will remain fully compatible with the previous version of the tool. All software components stay in their place like before, but some of them do not perform the same tasks anymore.

Some of the components involved in PEPS2015 software architecture, which can be observed in Figure 4.1, had their behavior changed, as well the data flow in some of the processes in order to achieve the objectives described earlier in this document.

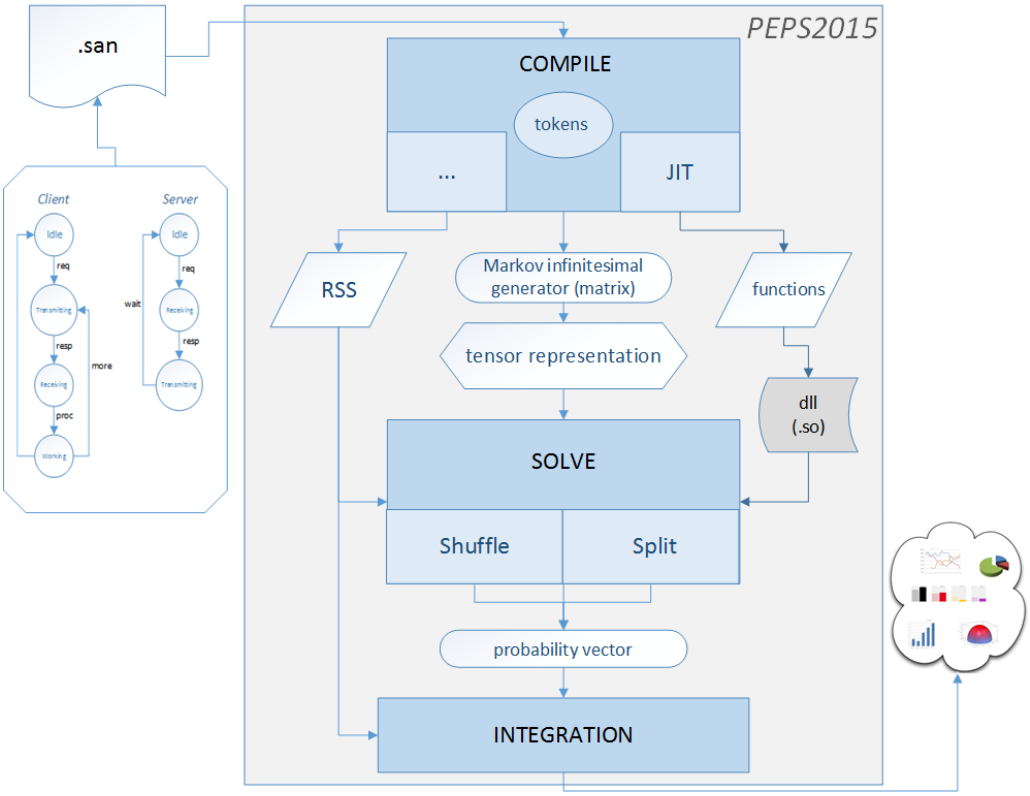

Figure 4.1 – PEPS2015 Architecture

First, the SAN model is described in a .san file according to the rules mentioned previously. Then PEPS takes the file as input and compiles each section of the file.

When parsing the file, it splits the content into tokens that go through the lexical, syntactic, and semantic analysis. These tokens are also used to store the functions present in the model. Until PEPS2007, they used to be stacked to represent the functions in a reverse-polish notation. From now on, the compilations of the functions is one of the software components that will be changed.

The Reachable State Space is currently calculated manually by using arrays of booleans to determine whether each state is reachable or not. The use of Multi-Valued Diagrams is yet to be implemented.

The Just-in-Time function evaluation is responsible for translating the functions defined in the model into C-coded functions that will be called in runtime. It produces a shared object to be linked with PEPS and generate the executable. The biggest change for the new version is focused in this specific part.

Once the model is compiled, we have the a tensor representation of the descriptor for the matrix that corresponds to the Markovian infinitesimal generator. This way, we only need to store the descriptor, instead of the entire matrix.

The solution module makes use of the reachable states, as well as the functions translated into C code by the jit module and applies the numerical iterative methods to solve the model based in the generated descriptor and the algorithm that is used, populating the probability vector as an output.

In the final step, the integration functions consider the values of the probability vector to produces the final results for the expected outputs that were specified for the model in order to calculate performance indexes.

4.2 Technology

In this section, we talk about the technology used in the development of the software tool:

4.2.1 Programming Language

Technology-wise, the software tool implements the communication between the internal modules via object oriented programming, using C++ programming language. Apart from that, basic input/output operations involving files and standard IO through Linux/Unix terminal are mplemented as well.

4.2.2 Environment

In regards to compatibility, PEPS package has been tested and proven to be currently compatible with the following platforms:

1. Linux - Ubuntu 14.04 64 bits
2. MacOS X 10.8.2 - Mountain Lion
3. MacOS X 10.9.2 - Mavericks
4. MacOS X 10.10.2 - Yosemite

5. MacOS X 10.11.1 - El Capitan

However, since the source code of PEPS is available upon request, the compilation and availability for other platforms is likely to be a simple task.

4.3 PEPS 2015 - New Features

The main contribution of this work relies on adding some new features to meet the objectives mentioned previously. In that sense the latest version of the software tool, PEPS2015, aims to allow models that have never been supported before because of stack issues to be supported now and improve total memory usage. We intend to get it done by working around the use of the stack to store the functional rates and getting rid of the temporary data structures also used for that purpose. A more detailed explanation is presented hereafter.

4.3.1 C Functions

Just-in-Time Functions

One of PEPS features that was implemented in the 2007 version is the so-called Just-in-Time functions. Basically, when the compiler parses the functions defined in the .san file for the functional transition rates in the model, it uses a Stack-based data structure to store all tokens involved to form each function. Then, once all functions are stored in the internal data structures, a *.jit* file is created, containing the translation of those functions into functions written in C programming language, *i.e.* , C-coded functions. The jit file is then linked to GCC to produce a shared object that will compound the executable file in a way that the C functions will be called whenever they need to be evaluated in runtime.

However, as described in Chapter 2, the way this feature is currently implemented creates some limitations for the tool capabilities. The program not ony uses a stack to store the tokens, but it also store them in temporary data structures, as shown in Figure 2.1.

More specifically, the program uses a function dictionary that store the valid token identifiers of the functions defined in the SAN model. Besides, it also uses a temporary list of tokens to control when the function ends, as tokens keep being processed one by one, so the program knows which token is the final one for that function. There is also a table of *function trees*. A function tree is a list of tokens that compound one function of the model. So, the table contains a list of all functions of the model in the form of token lists.

The functions are actually translated into C code and sent to the jit file only after the table of function trees contains all of the functions present in the model. That takes a large amount of memory in compile time and, more importantly, the fact that tokens are stacked up to form

functions implies that the amount of functions in the model and the size of those functions will determine if the model can be handled by the tool or not, for it depends on the size of the stack the program uses to store them.

Proposed solution

The proposed solution intends to change the way the compiler parses functional rates and stores them internally to generate the jit file. It is also expected that some additional benefits will come along with the implemented solution.

First of all, since the JIT module in PEPS architecture ultimately generates C coded functions from the original functions defined in the SAN model, we thought it would be more reasonable to let the modeler describe the functional rates in the SAN file using C programming language directly. This way, not only all of the intermediate translation steps would no longer be necessary, but it would also allow the modeler to write his/her own algorithms, using C language resources, such as sort algorithms, etc. With that in mind, of the expected additional benefts is facilitate the modeling process, regarding function design, and the understanding of the functions defined for the model.

For example, it is much easier for the modeler to use a *switch-case* clause from C language instead of specifying each and every single condition with *if* statements, as it is currently defined in the .san file. Additionaly, when the function is evaluated, only one verification will be performed with the *switch-case* clause, instead of *n* verifications for *n if* statements, therefore even some minor performance improvements are expected.

In order to make it work, the internal structure of PEPS has to be slightly changed. Since the functions will be written in C language, we need to modify the compiler. So, some elements were added to enable PEPS compiler to recognize C code, such as a new grammar for C specific symbols, as well as a new parser with syntactic and semantic rules. It is important to point that the new compiler was based on the compiler implemented in the 2007 version, without using Lex/Yacc. Additionally, the .san file now has a new section that comes right after the *identifiers* section, called *functions*. This is where the new C functions will be coded in order to be recognized by the compiler.

Once the compiler is now able to parse proper C code directly from the input file, we needed to find a way of dealing with the C functions internally in order to create the jit file just as it was done before. We then realized that since we already have C code, there is no need to use any temporary data structure to do any sort of translation process. We could just froward the C code to an output stream that, once all functions are copied, could be written directly in the jit file.

It seemed like a good approach, but there was one issue. Even though the function code was written in C language, the conditions would still have to check the automata states, and obviously there is nothing in C language to do that. So, we had to find a way of interpreting some SAN-specific symbols in the middle of the C code to resolve that issue. Then, those specific

symbols from the SAN grammar were added to the new compiler and the correct information about the current state of each automata would come from acessing the *network* dictionary.

The network dictionary is one of PEPS internal data structures that store data about the structure of the model itself. When the *network* section of the *.san* input file is parsed, this dictionary is filled with all relevant information regarding the network structure of the model (description of automata, states, and transitions).

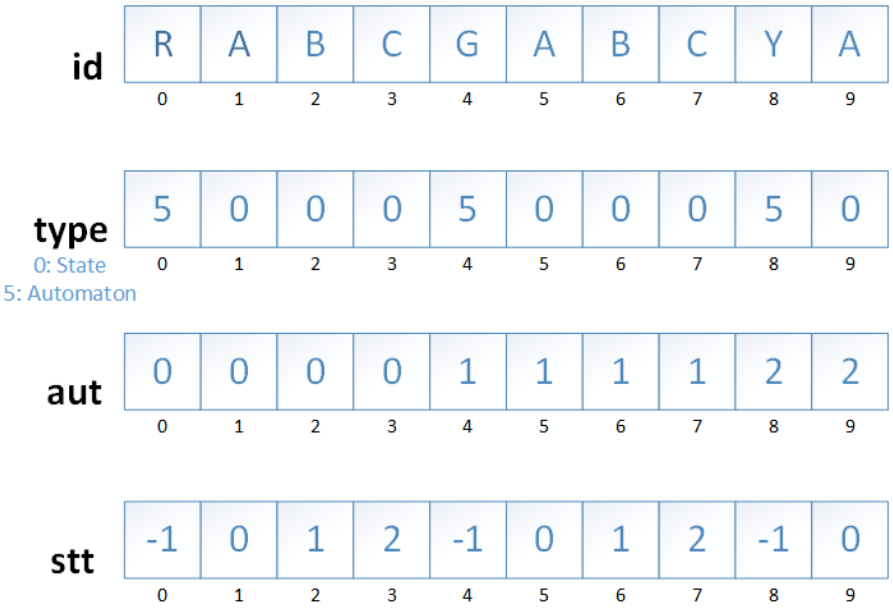

Figure 4.2 – Network dictionary

Basically, the network dictionary is composed of four lists to store the model, as shown in Figure 4.2:

- **id**: Each position of this list stores the name of an identifier.
- **aut**: Each position of this list stores the index of the automaton that the identifier belongs to.
- **type**: Each position of this list indicates the type of the identifier, whether it is an automaton identifier or a state identifier.
- **stt**: Each position of this list indicates the index of the state inside the automaton that state belongs to. For automaton identifiers, the value is disregarded.

For the sake of the example, Figure 4.2 contains a model with three automata, named *R, G*, and *Y*, each with three states, named *A, B* and *C*, being state *A* indexed as *0*, state *B* indexed as *1*, and state *C* with index *2* inside each automaton.

So, by acessing the network dictionary, it was possible to resolve the issue with specific SAN grammar elements. To make it clearer for the reader, here is another example:

Let's assume a function called *reset*, that triggers an event in a transition only *if the state of automaton* A *is* 1 *and the state of automaton* B *is* 1.

The SAN function, as defined in PEPS2007 would be:

$$\textit{reset} = ((\textit{st A} == 1)\&\&(\textit{st B} == 1)) * 1$$

In PEPS2015, the C code for that function would be:

$$\textit{double reset}(< \textit{parameters} >)\{\ \textit{if}((A == 1)\&\&(B == 1))\ \textit{return}\, 1;\ \textit{else return}\, 0\}$$

This is only meant to exemplify how SAN grammar specific elements that do not exist in C syntax would have to be dealt with, such as ***st*** and automata names, in this case *A* and *B*. The symbol *st* was added to the grammar and both the automata names and the values of the states being checked would have to be searched on the network dictionary.

So, by searching for the values on the same index in all four lists contained in the dictionary, it is possible to know if an identifier is valid, what kind of identifier we are dealing with, which automaton it belongs to, and, in case it's a state, the index of that state in the automaton.

This way, it became straightforward to generate a new jit file with the same C functions as before, but using a much faster and simpler process. However, there was still one remaining issue.

Since the final output of the tool is given in the form of probabilities, all of the values calculated during the execution of the program have to be normalized, *i.e.* , transformed into equivalent values within the $[0.0, 1.0]$ range. In that sense, one of the calculated values during the process is the *normalizing factor*. It is stored in one of the files generated by the compilation process and it must be applied to the functions in order to work properly.

This is why the program has to parse the input file more than once. In the first pass, it stores the identifiers present in the model and the structure of the network itself. The second time it passes through the model, the network structure is already mapped internally and the necessary values are calculated, such as the normalizing factor. Since the network structure has to be stored internally to process the functional rates associated to transition events between states properly, a third pass was added to the compiling process, which is when the C functions are actually parsed, containing all necessary values.

The files created by the compiling process are:

- DES: stores all information concerning the network structure.
- DSC: stores the internal descriptor of the model.
- CND: stores the normalized descriptor (contains the normalizing factor).

So, in order to resolve the normalizing factor issue, we had to generate a partial jit file, with not-normalized functions. It was called *not-normalized jit file* and it contains all of the C functions with an invalid token (character "**?**") for the value of the normalizing factor. This not-normalized jit

file is then created right before the creation of those three files mentioned above, once the compiling process is over.

After those files are all created, we know the normalized factor was already calculated and stored internally in the tool. So, we could reopen the not-normalized jit file, find the invalid token (question mark symbol) intentionally put there as the normalizing factor, and finally replace it with the actual value of the normalizing factor calculated for the model, to which we have access at this point because it was calculated during the compilation process. This way, a second and final jit file, called *normalized jit*, was generated to replace the old one and still work the same way.

GCC is called to compile the new jit file and produce the shared object to be linked with PEPS and generate the executable module. Function evaluation is still done as efficiently as before.

By making these modifications, we were able to meet the goal of not using the stack anymore to store functions, because we no longer use token lists or function trees. The C code is directly parsed and saved to a stream that is later on written in the jit file. These adjustments in the structure and data flow also eliminated the need of using temporary data structures to store the tokens and build the functions. So, it is expected that models that were not supported before due to stack limitation or memory overflow issues can now be handled by the software tool.

4.3.2 Example of the new .san file

In this example, we are modeling the Tower of Hanoi problem with three discs. There are three automata, equivalent to the discs, R for the the red and smallest disc, G for the green medium disc, and Y for the yellow biggest disc. Each of these automata have three states, A, meaning that the disc is inserted in the left pole A, B meaning the disc is in the middle pole B, and C that stands for the right pole C.

Basically, the three discs start on the top of each other, biggest at the bottom, medium and smallest on the top, all inserted into the left pole A. Only the disc on the top can be switched from one pole to another at a time. The goal is to move all three discs from pole A to pole C without ever putting a disc on the top of a smaller disc.

Figure 4.3 depicts the scenario explained above.

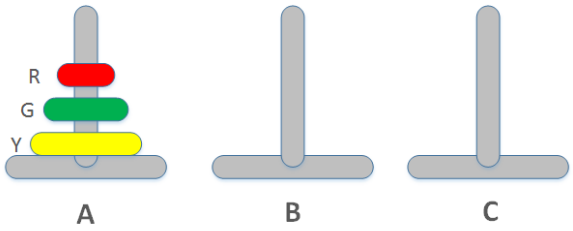

Figure 4.3 – Example of Hanoi Tower

Figures 4.4 and Code 4.1 show examples of the SAN model for the Hanoi Tower example and the .san file that describes it in the new format, respectively, with the proposed modifications.

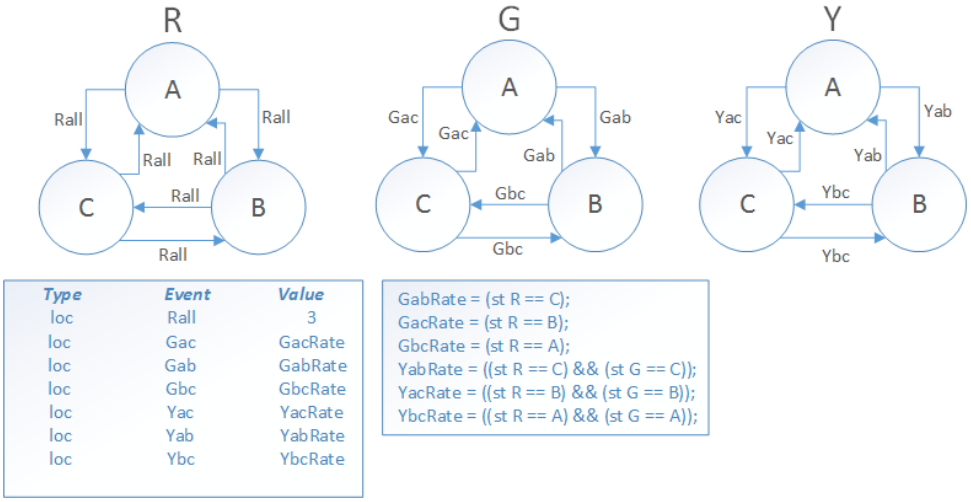

Figure 4.4 – Example of Hanoi Tower - SAN model

```
identifiers

  RallRate = 3;
  GabRate = (st R == C);
  GacRate = (st R == B);
  GbcRate = (st R == A);
  YabRate = ((st R == C) && (st G == C));
  YacRate = ((st R == B) && (st G == B));
  YbcRate = ((st R == A) && (st G == A));

functions

double GacRate(const int * states){
  double res;
  double norm_factor = ?;
  if(st R == B){
    res = 1;
  }
  else{
    res = 0;
  }
  res = res * norm_factor;
  return res;
}

double GabRate(const int * states){
  double res;
  double norm_factor = ?;
  if(st R == C){
    res = 1;
  }
  else{
    res = 0;
  }
  res = res * norm_factor;
  return res;
}
```

```
double GbcRate(const int * states)
{
  double res;
  double norm_factor = ?;
  if(st R == A)
    res = 1;
  else
    res = 0;
  res = res * norm_factor;
  return res;
}

double YacRate(const int * states)
{
  double res;
  double norm_factor = ?;
  if((st R == B) && (st G == B))
  {
    res = 1;
  }
  else
  {
    res = 0;
  }
  res = res * norm_factor;
  return res;
}

double YabRate(const int * states)
{
  double res;
  double norm_factor = ?;
  if((st R == C) && (st G == C))
  {
    res = 1;
  }
  else
  {
    res = 0;
  }
  res = res * norm_factor;
  return res;
}

double YbcRate(const int * states)
{
  double res;
  double norm_factor = ?;
  if((st R == A) && (st G == A))
  {
    res = 1;
  }
  else
  {
    res = 0;
  }
  res = res * norm_factor;
  return res;
}
```

```
events

  loc Rall  (RallRate);
  loc Gac (GacRate);
  loc Gab (GabRate);
  loc Gbc (GbcRate);
  loc Yac (YacRate);
  loc Yab (YabRate);
  loc Ybc (YbcRate);

reachability = ((st R == A) && (st G == A) && (st Y == A));

network hanoi (continuous)

  aut R
    stt A to (B)  Rall
          to (C)  Rall
    stt B to (A)  Rall
          to (C)  Rall
    stt C to (A)  Rall
          to (B)  Rall

  aut G
    stt A to (B)  Gab
          to (C)  Gac
    stt B to (A)  Gab
          to (C)  Gbc
    stt C to (A)  Gac
          to (B)  Gbc

  aut Y
    stt A to (B)  Yab
          to (C)  Yac
    stt B to (A)  Yab
          to (C)  Ybc
    stt C to (A)  Yac
          to (B)  Ybc

results
  t_A = ((st R == A) && (st G == A) && (st Y == A));
  t_B = ((st R == B) && (st G == B) && (st Y == B));
  t_C = ((st R == C) && (st G == C) && (st Y == C));
```

Code 4.1 – Example of the SAN file for Hanoi tower model

5. PRACTICAL CAPABILITIES

5.1 Analysis Criteria

Based on the analysis criteria defined before to test the previous versions capabilities, the same model tested in PEPS2007 version was tested in the new version to be sure that the new implementation works and verify eventual performance improvements. The results are presented in Table 5.1

Table 5.1 – PEPS2015 numerical results

PEPS version	Model	PSS	Memory Used	Time to Solve
2000	General Resource Sharing [17]	1,084,576	590 Bytes	33 sec.
2003	FAS - First Server Available [8]	33,554,432	11.2 KB	5.47 min.
2007	Master-Slave Parallel Program [6, 14]	65,367,200	38.54 KB	8.92 min.
2015	Master-Slave Parallel Program [6, 14]	65,367,200	29.12 KB	8.84 min.

As it can be observed, the new implementation of the tool did reduce the total amount of memory used, as expected. We believe that the reduction of approximately 24.44% is mostly because of the new compiling process and the runtime should still be using about the same amount of memory as before. The results also show that the new implementation still supports previously supported models.

5.2 Test Scenarios

5.2.1 Hanoi Tower

In this setion, we present the tests done on the old and new versions of the Hanoi Tower scenario shown before. The scenario was modeled for PEPS2007 using the old grammar and also for PEPS2015 with the new format. The model was then executed in both versions and the expected results are the same stationary solution for both models, indicating that the numeric algorithms can still solve the model the same way, using the new C functions defined by the modeler in project time.

The compilation and execution of both models are shown in Figures 5.1, 5.2, 5.3, and 5.4.

```
    +-------------------------------------------------------+
    |            This is PEPS 2007 - the SAN tool           |
    |  released on: August 2015 -- compiled on: Nov 16 2015  |
    +-------------------------------------------------------+

  1) Compile a SAN model             5) Load/Show/Remove data structures
  2) Solve a compiled SAN model      6) Inspect data structures
  3) Probability vector facilities   7) Sparse matrix facilities
  4) Preferences                     8) About this version

     0) Exit PEPS (Option 0 always exits the current menu)
1


             ******* Compiling a SAN Model *******

 1) Standard Compilation (PEPS 2003 version)
 2) Compilation with Algebraic Aggregation
 3) Compilation with Aggregation of Replicas
 4) Compilation of PEPS 2007 version
4

Enter textual SAN file name: hanoi_old
Start model compilation
...
:-) file 'cnd/hanoi_old.cnd' saved
:-) file 'cnd/hanoi_old.ftb' saved
:-) file 'cnd/hanoi_old.rss' saved
:-) file '/home/lgz/workspace/peps/jit/peps_jit.C' saved
```

Figure 5.1 – Compilation of the Hanoi Tower SAN model in PEPS2007

```
    +-------------------------------------------------------+
    |            This is PEPS 2007 - the SAN tool           |
    |  released on: August 2015 -- compiled on: Nov 16 2015  |
    +-------------------------------------------------------+

  1) Compile a SAN model             5) Load/Show/Remove data structures
  2) Solve a compiled SAN model      6) Inspect data structures
  3) Probability vector facilities   7) Sparse matrix facilities
  4) Preferences                     8) About this version

     0) Exit PEPS (Option 0 always exits the current menu)
2


       ******* Solving a SAN model *******

No Preconditioning:      Diagonal Preconditioning:

 1) Power Method             4) Power Method
 2) Arnoldi Method           5) Arnoldi Method
 3) GMRES Method             6) GMRES Method

 7) Uniformization Method (transient solution)

 8) Symbolic Solution Method
1

Solution of the model 'cnd/hanoi_old.cnd' (3 automata - 1/27 states)

Enter vector file name: hanoi old res
...
Iteration 533:
Power solution
Number of iterations:   534
 - user time spent:     1.3674000000000000e-02 seconds
 - system time spent:   0.0000000000000000e+00 seconds
 - real time spent:     1.2625932693481445e-02 seconds
Residual Error: 9.6047572673008119e-11 - The method converged (solution found)!
 :-) file 'hanoi_old_res.vct' saved
 :-) file 'hanoi_old.tim' saved

Integration (dsc/hanoi_old.inf) of the vector 'hanoi_old_res.vct' (27 states) solution of 'cnd/hanoi_old.cnd' (27 states)
Integration of function t_A : 3.7037040014364420e-02
Integration of function t_B : 3.7037035548373322e-02
Integration of function t_C : 3.7037035548373322e-02
```

Figure 5.2 – Resolution of the Hanoi Tower SAN model in PEPS2007

5.2.2 Discrete Spatial Mobile Node Distribution

```
    +-------------------------------------------------------+
    |            This is PEPS 2015 - the SAN tool           |
    |  released on: August 2015 -- compiled on: Nov 16 2015  |
    +-------------------------------------------------------+

  1) Compile a SAN model             5) Load/Show/Remove data structures
  2) Solve a compiled SAN model      6) Inspect data structures
  3) Probability vector facilities   7) Sparse matrix facilities
  4) Preferences                     8) About this version

     0) Exit PEPS (Option 0 always exits the current menu)
1


            ******* Compiling a SAN Model *******

 1) Standard Compilation (PEPS 2003 version)
 2) Compilation with Algebraic Aggregation
 3) Compilation with Aggregation of Replicas
 4) Compilation of PEPS 2015 version
4

 Enter textual SAN file name: hanoi_new
 Start model compilation
 ...

 :-) file '/home/lgz/workspace/peps/jit/peps15_jit.C' saved
 :-) file '/home/lgz/workspace/peps/jit/peps15_jit_notnorm.C' read
 :-) file 'cnd/hanoi_new.cnd' saved
 :-) file 'cnd/hanoi_new.ftb' saved
 :-) file 'cnd/hanoi_new.rss' saved
```

Figure 5.3 – Compilation of the Hanoi Tower SAN model in PEPS2015

One of the models that users have been trying to solve for a long time using PEPS software tool is the Discrete Spatial Mobile Node Distribution [16]. It consists in finding a suitable mobility model for packages to travel across computer wireless networks. In other words, the main problem to be resolved in this model is finding an optimal path for the signal to travel from one waypoint to another given a spatial node distribution. More details about the model problem itself can be found in [16].

The reason why this model could not be supported so far is the exactly problem identified in this work, which we intended to solve. Because of the very large number of complex functions that were modeled to represent this scenario, the stack would overflow whenever the model was attempted to be solved.

With the new implementation, this model is very likely to be supported by the 2015 version. Unfortunately, we could not execute the stochastic model in SAN to PEPS2015 because the model itself was not published, only the description of the problem itself is available. But, according to the authors, it becomes unsolvable when the Spatial Node Distribution is done in a 20x20 area with varied pauses on transmission, forming a non-convex polygon. This would be the desired test for the new implementation, but it is estimated that the model is highly likely to be supported by PEPS2015 version.

A truly useful and important future work that can be done on this version is modeling this scenario and confirm whether it works indeed or it still remains unsupported due to memory limitations.

```
     +-----------------------------------------------------+
     |            This is PEPS 2015 - the SAN tool         |
     |  released on: August 2015 -- compiled on: Nov 16 2015  |
     +-----------------------------------------------------+

  1) Compile a SAN model             5) Load/Show/Remove data structures
  2) Solve a compiled SAN model      6) Inspect data structures
  3) Probability vector facilities   7) Sparse matrix facilities
  4) Preferences                     8) About this version

     0) Exit PEPS (Option 0 always exits the current menu)
2


       ******* Solving a SAN model *******

No Preconditioning:      Diagonal Preconditioning:

 1) Power Method             4) Power Method
 2) Arnoldi Method           5) Arnoldi Method
 3) GMRES Method             6) GMRES Method

 7) Uniformization Method (transient solution)

 8) Symbolic Solution Method
1

Solution of the model 'cnd/hanoi_new.cnd' (3 automata - 1/27 states)
...
Iteration 533:
Power solution
Number of iterations:   534
 - user time spent:     5.2709999999999996e-03 seconds
 - system time spent:   2.9019999999999983e-03 seconds
 - real time spent:     9.6499919891357422e-03 seconds
Residual Error: 9.6047572673008119e-11 - The method converged (solution found)!
 :-) file 'hanoi_new_res.vct' saved
 :-) file 'hanoi_new.tim' saved

Integration (dsc/hanoi_new.inf) of the vector 'hanoi_new_res.vct' (27 states) solution of
Integration of function t_A : 3.7037040014364420e-02
Integration of function t_B : 3.7037035548373322e-02
Integration of function t_C : 3.7037035548373322e-02
```

Figure 5.4 – Resolution of the Hanoi Tower SAN model in PEPS2015

5.3 Results and Validation of Implemented Features

According to the output shown in Figures 5.1, 5.2, 5.3, and 5.4, we were eable to validate the implemented features, since the resolution process still achieves the stationary solution for the model, using the new just-in-time functions.

The results presented in 5.1 also confirm the new implementation is fairly acceptable and the expected benefits for the tool were obtained. Additionaly, as a positive side effect, we have made the modeling process simpler and more intuitive for the user of PEPS tool.

6. FUTURE WORK

6.1 Symbolic Solution

One significant feature that could be added to the software tool in the future is the Symbolic Solution, described in [18]. According to the authors, by making use of tensor algebra principles, it is expected that much larger and complex models would be resolved without increasing the execution time and memory usage all that much.

The main concept of the symbolic solution is to deal with the tensor representation of the infinitesimal generator by adding terms to perform Gauss-Jordan Elimination steps. The concept of functional elements is highly important for the method to deal with particular cases of the tensor structure and keep the symbolic operations simple enough to be handled.

The first step for the proposed symbolic solution is to prepare the matrix by performing the Gauss-Jordan method to obtain the infinitesimal generator. Once this process is done, the tensor representation of the starting of the Gauss-Jordan method will be:

$$Q^{(0)} = \bigoplus_{i=0}^{N-1} Q_l^{(i)} + \sum_{s \in S} \bigotimes_{i=0}^{N-1} Q_{s^+}^{(i)} + \sum_{s \in S} \bigotimes_{i=0}^{N-1} Q_{s^-}^{(i)} + \bigotimes_{i=0}^{N-1} Q_{lc}^{(i)} \tag{6.1}$$

The second step is to perform the matrix triangulation, in which the elements below the diagonal are eliminated through he application of the Gauss-Jordan to each row. The row scaling operations correspond to include functional elements to perform the necessary scale operation to each product of the descriptor. Once the matrix triangulation is complete, it is computationally simpler to store the tensor structure and execute the final step: perform a backward substitution procedure to resolve the system of linear equations. More detailed information about the method itself is presented by Fernandes, Lopes and Yeralan in [18].

6.1.1 Meaningful Test Scenario

For future implementations, one model that is known to have a very large state space and could be tested to look for performance improvements is the *Unreliable Production Lines* model [19].

Also, as mentioned before, Chapter 3.6.1 describes one important feature to calculate the reachable state space of the model in a more sophisticated way. In Chapter 5.2.2, another meaningful test scenario is presented, for it could be used to validate the improvements on the tool made by this project.

7. CONCLUSION

Fig. 1.1 summarizes the general vision of the PEPS2015 software high level modules, as well as types of data and data flow involved in the compiling/solving process. In the figure, the left hand side shows the input of PEPS 2015, a textual .san file describing a specific SAN model, and the right hand side shows the final output: the results computed by applying the integration functions to the computed probability vector, *i.e.* , the quantitative estimations concerning the modeled reality. The three main modules of PEPS 2015 are the Compile, Solve and Integrate modules.

The *Compile* module performs a translation of a SAN model into an MDD description of the Reachable State Space, a tensor representation of transition matrix (descriptor), and the integration functions responsible to deliver the quantitative results. It comprises two specialized submodules, one responsible for generating the MDD representation of the RSS, and the other responsible for generating the *just-in-time* (jit) functions for the tensor representation and the integration functions.

The *Solve* module implements all iterative numeric methods available in the tool to try to solve the model, namely *Power, Arnoldi, GMRES* and *Uniformization*, offering Shuffle and Split algrithms for vector-descriptor multiplications. This module can be seen as the most sensitive module in terms of computational efficiency, in terms of both CPU and memory usage, since it performs the operations for achieving the stationary solution.

Finally, the *Integration* module is the simplest and most straightforward module, since it essentialy visits the Reachable State Space, computing the integration functions and weighing results from the probabilities computed beforehand.

PEPS software tool is a live project, hence there is constantly ongoing work to add new features and keep improving it by seeking for even better techniques for vector-descriptor multiplication to speed up both transient and stationary numerical solutions, as well as more sophisticated structures for reachable state space manipulation. The new features proposed in this work led to a new version of PEPS as the core contribution for the project.

At the end of this project, we appreciate all the acquired knowledge about the subject. We had the chance of going deep into the details of analytical modeling and explore the methods and techniques that have been developed along the years by talented researchers in the past.

We were satisfied with the results because the initial objectives that had been set were achieved, improving the software tool usage and execution and hopefully opening up some opportunities for even better future implementations.

The contribution for the academic community is something truly valuable for us and the research pratice performed along this project was worth the effort. The article published in the 31st UK Performance Engineering Workshop, held in 17 September 2015 in Leeds [22], is the clearest proof of it. We are glad to be part of the scientific community and wll surely keep working hard to contribute even more in the future.

As new techniques are proposed in the future, they may result in newer and optimized versions of PEPS tool. The interested researcher and practitioner may find more thorough information about PEPS project in general in the webpage http://www-id.imag.fr/Logiciels/peps/.

REFERENCES

[1] "J. J. O'Connor and E. F. Robertson. MacTutor History of Mathematics., http://www-gap.dcs.st-and.ac.uk/history/mathematicians/markov.html, Accessed: October, 2015".

[2] "KRONOS, http://www-verimag.imag.fr/dist-tools/tempo/kronos/, Accessed: May, 2015".

[3] "Lex - A Lexical Analyzer Generator, http://dinosaur.compilertools.net/lex/, Accessed: September, 2015".

[4] "UPPAAL, http://www.uppaal.org/, Accessed: May, 2015".

[5] "Yacc: Yet Another Compiler-Compiler, http://dinosaur.compilertools.net/yacc/, Accessed: September, 2015".

[6] Baldo, L.; Brenner, L.; Fernandes, L. G.; Fernandes, P.; Sales, A. "Performance Models For Master/Slave Parallel Programs", *Electronic Notes in Theoretical Computer Science*, vol. 128–4, 2005, pp. 101 – 121, proceedings of the First International Workshop on Practical Applications of Stochastic Modelling (PASM 2004) Practical Applications of Stochastic Modelling 2004.

[7] Benoit, A.; Brenner, L.; Fernandes, P.; Plateau, B. "Aggregation of Stochastic Automata Networks with Replicas", *Linear Algebra and its Applications*, vol. 386, 2004, pp. 111–136.

[8] Benoit, A.; Brenner, L.; Fernandes, P.; Plateau, B.; Stewart, W. "The peps software tool" . In: *Computer Performance Evaluation. Modelling Techniques and Tools*, Kemper, P.; Sanders, W. (Editors), Springer Berlin Heidelberg, 2003, *Lecture Notes in Computer Science*, vol. 2794, pp. 98–115.

[9] Brenner, L. "Agregação em Redes de Autômatos Estocásticos", Ph.D. Thesis, Pontifícia Universidade Católica do Rio Grande do Sul, 2004.

[10] Brenner, L.; Fernandes, P.; Plateau, B.; Sbeity, I. "PEPS2007 - Stochastic Automata Networks Software Tool." In: QEST, 2007, pp. 163–164.

[11] Bujorianu, L. M. "Stochastic reachability analysis of hybrid systems". Springer Science & Business Media, 2012.

[12] Ciardo, G.; Lüttgen, G.; Miner, A. S. "Exploiting Interleaving Semantics in Symbolic State-Space Generation", *Formal Methods in System Design*, vol. 31–1, 2007, pp. 63–100.

[13] Ciardo, G.; Lüttgen, G.; Siminiceanu, R. "Saturation: an Efficient Iteration Strategy for Symbolic State—Space Generation". Springer, 2001.

[14] Czekster, R. M.; Fernandes, P.; Vincent, J.-M.; Webber, T. "Split: a flexible and efficient algorithm to vector-descriptor product". In: Proceedings of the 2nd international conference on Performance evaluation methodologies and tools, 2007, pp. 83.

[15] Czekster, R. M.; Fernandes, P.; Webber, T. "Efficient Vector-Descriptor Product Exploiting Time-Memory Trade-offs", *ACM SIGMETRICS Performance Evaluation Review*, vol. 39–3, 2011, pp. 2–9.

[16] Dotti, F. L.; Fernandes, P.; Nunes, C. M. "Structured markovian models for discrete spatial mobile node distribution", *Journal of the Brazilian Computer Society*, vol. 17–1, 2011, pp. 31–52.

[17] Fernandes, P. "Méthodes numériques pour la solution de systèmes Markoviens à grand espace d'États", Ph.D. Thesis, Institut National Polytechnique de Grenoble, France, 1998.

[18] Fernandes, P.; Lopes, L.; Yeralan, S. "Symbolic Solution of Kronecker-based Structured Markovian Models". In: Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2013 IEEE 21st International Symposium on, 2013, pp. 409–413.

[19] Fernandes, P.; O'Kelly, M.; Papadopoulos, C. T.; Sales, A. "Analysis of Exponential Unreliable Production Lines using Kronecker Descriptors", *9th Stochastic Models of Manufacturing and Service Operations (SMMSO 2013)*, 2013.

[20] Fernandes, P.; Plateau, B.; Stewart, W. J. "Efficient Descriptor-Vector Multiplication in Stochastic Automata Networks", *Journal of the ACM (JACM)*, vol. 45–3, 1998, pp. 381–414.

[21] Fernandes, P.; Presotto, R.; Sales, A.; Webber, T. "An Alternative Algorithm to Multiply a Vector by a Kronecker Represented Descriptor". In: 21st UK Performance Engineering Workshop, 2005, pp. 57–67.

[22] Fernandes, P.; Sales, A.; Zani, L. "Peps2015-stochastic automata networks software tool". In: 31st UK Performance Engineering Workshop 17 September 2015, 2015, pp. 9.

[23] Fourneau, J.-M.; Plateau, B.; Sbeity, I.; Stewart, W. "Sans and lumpable stochastic bounds: Bounding availability", *Computer System, Network Performance and Quality of Service*, 2004, pp. xxxx.

[24] Kam, T.; Villa, T.; Brayton, R. "Multi-Valued Decision Diagrams: Theory and Applications", *Multiple-Valued Logic*, –4, 1998.

[25] Kleinrock, L. "Queueing Theory". John Wiley, 1979.

[26] Langville, A. N.; Stewart, W. J. "The Kronecker Product and Stochastic Automata Networks", *Journal of computational and applied mathematics*, vol. 167–2, 2004, pp. 429–447.

[27] Marsan, M. A.; Balbo, G.; Conte, G.; Donatelli, S.; Franceschinis, G. "Modelling with generalized stochastic Petri nets". John Wiley & Sons, Inc., 1994.

[28] Miner, A. S.; Ciardo, G. "Efficient Reachability Set Generation and Storage Using Decision Diagrams". In: *Application and Theory of Petri Nets 1999*, Springer, 1999, pp. 6–25.

[29] Plateau, B.; Fourneau, J.-M.; Lee, K.-H. "PEPS: A package for solving complex Markov models of parallel systems". Springer, 1989.

[30] Plateau, B.; Stewart, W. J. "Stochastic Automata Networks". In: *Computational Probability*, Springer, 2000, pp. 113–151.

[31] Saad, Y. "Iterative Methods for Sparse Linear Systems". Siam, 2003.

[32] Sales, A.; Plateau, B. "Reachable State Space Generation for Structured Models Which Use Functional Transitions". In: Quantitative Evaluation of Systems, 2009. QEST'09. Sixth International Conference on the, 2009, pp. 269–278.

[33] Stewart, W. J. "Numerical Solution of Markov Chains". vol. 8.

[34] Uysal, E.; Dayar, T. "Iterative Methods Based on Splittings for Stochastic Automata Networks", *European Journal of Operational Research*, vol. 110–1, 1998, pp. 166–186.