MAGNET: a Software Library for Markovian Agent Networks

Marco Scarpa
mscarpa@unime.it
Department of Engineering
University of Messina
Messina, Italy

Giuseppe Molica
gmolica17@gmail.com
Department of Engineering
University of Messina
Messina, Italy

ABSTRACT

Continuous Time Markov Chains (CTMC) are a mathematical tool widely used in many modeling areas. Main drawback in using CTMCs is the difficulty in building and managing models with a high number of states. In [3] a new modeling technique based on Markovian Agents has been proposed to study Wireless Sensor Networks with a large number of interacting nodes. A Markovian Agent is a CTMC with the ability to interact with other Markovian Agents by sending and perceiving messages. Thanks to message mechanisms, it is possible to limit the growth of the overall state space. In this paper, we introduce a software to implement Markovian Agent models in a simple way. Software functionalities are provided through a C library we named *MAGNET* (**M**arkovian **AG**ent **NET**works). It provides a simple interface to easily build complex interacting MA models and a numerical algorithm to study them in transient exploiting multi-threading.

CCS CONCEPTS

• **Computing methodologies** → **Model development and analysis**; **Modeling methodologies**; • **Software and its engineering** → *Software libraries and repositories*;

KEYWORDS

Continuous Time Markov Chain, Markovian Agents, C library, Performance Evaluation

ACM Reference Format:
Marco Scarpa and Giuseppe Molica. 2017. MAGNET: a Software Library for Markovian Agent Networks . In *11th EAI International Conference on Performance Evaluation Methodologies and Tools.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3150928.3150955

1 INTRODUCTION

Continuous Time Markov Chains (CTMCs) are widely used in performance and reliability modeling to study systems with complicated interactions among components [11]. Notion of space becomes relevant in most systems whose behavior is directly dependent on the environment. Many systems take advantage from modeling formalism able to represent object locations. During last years a lot of effort has been devoted to improve modeling techniques with notion of space [1, 2, 4, 6, 10]. Anyway analysis cost of such kind of models is very high due to the state space explosion problem.

Markovian Agents modeling paradigm, originally proposed to study Wireless Sensor Networks with a large number of interacting nodes [3], tries to face state space explosion problem taking into account discrete state space where CTMCs could be located. Each spatially located CTMC, referred to as *Markovian Agent* (*MA*), can interact with other *MA*s by exploiting message exchange mechanism. Each agent maintains its local properties, but at the same time modifies its behavior according to the influence of interactions with other agents. In this way, analysis of each agent alone incorporates the effect of the inter-dependencies and is performed by referring to its own state space; the final result is that it is not needed to build the overall system state space as the cross product of each MA state space.

In this paper, we introduce a software framework implementing all the functionalities to build and evaluate a Markovian Agent Model in an easy way. Our work has been inspired by the successful story of *SPNP* [5] and *SHARPE* [8] in the field of Generalized Stochastic Petri nets, that originally are provided as C library and a programming environment, respectively, through which a modeler can specify his/her model by using a programming language. Similarly, we designed a C software library, we named ***M**arkovian **AG**ent **NET**works* (*MAGNET*) library[1], that allows the easy specification of a complete *Multiple Agent Class Multiple Message Type Markovian Agent Model* (M^3AM) and transient state probability computations of individual agents. At the best of our knowledge, no other software tools exist for implementing and studying M^3AM models.

The paper is organized as follows. Section 2 recalls Markovian Agent model definition and the equations used to compute transient probabilities. Section 3 introduces the MAGNET architecture. The multi-threaded algorithm implemented in MAGNET is described in Section 4. An example is discussed in Section 5, whereas Section 6 concludes the paper.

2 MARKOVIAN AGENT MODEL DEFINTION

Since its introduction in [3, 4], Markovian Agents Models (*MAM*s) have been defined as a collection of interacting continuous time Markov chains. Each CTMC in a *MAM* defines a *Markovian Agent* class, according to the usually used terminology [2–4]. In the following, we denote with the term *Markovian Agent (MA)* an instance of a Markovian Agent class. Each agent MA^c of class c is characterized by a state space with n_c states, and it is defined by the tuple:

$$MA^c = \{\mathbf{Q^c}, \mathbf{\Lambda}^c, \mathbf{G}^c(m), \mathbf{A}^c(m), \boldsymbol{\pi}_0^c\}. \quad (1)$$

[1] http://perf.unime.it/magnet

VALUETOOLS 2017, December 5–7, 2017, Venice, Italy

ACM ISBN 978-1-4503-6346-4/17/12.
https://doi.org/10.1145/3150928.3150955

Name	Symbol	Type	Multiplicity
Class	c	Label	1
Local transitions	$\mathbf{Q}^c$	Matrix	1
Self loops	$\mathbf{\Lambda}^c$	vector	1
Message generation probabilities	$\mathbf{G}^c(m)$	set of matrices	M
Message acceptance probabilities	$\mathbf{A}^c(m)$	set of matrices	M
Initial state probabilities	$\boldsymbol{\pi}_0^c$	vector	1

Table 1: Needed quantities for MA class declaration

$\mathbf{Q}^c = [q_{ij}^c]$ is the $n_c \times n_c$ infinitesimal generator matrix of the CTMC that describes the local behavior of a class c agent.

$\mathbf{\Lambda}^c = [\lambda_i^c]$, is a vector of size n_c storing the rates of *self-loops* for a class c agent.

$\mathbf{G}^c(m) = [g_{ij}^c(m)]$ is a $n_c \times n_c$ matrix describing the probability that an agent of class c generates a message of type m during a jump from state i to state j.

$\mathbf{A}^c(m) = [a_{ij}^c(m)]$ is a $n_c \times n_c$ matrix, that describes the acceptance probability of type m messages for an agent of class c.

$\boldsymbol{\pi}_0^c$, is the initial state probability vector of an agent of class c.

Table 1 synthesizes the quantities characterizing a class together with the type of each involved object (**Type)**, its multiplicity inside class definition (**Multiplicity**) and the symbols we use to denote them. M denotes the number of messages in the model.

When classes are declared, they can be used to define a model according the following definition. Formally a *Multiple Agent Class, Multiple Message Type Markovian Agents Model* (M^3AM) is defined by the tuple:

$$M^3AM = \{C, \mathcal{M}, \mathcal{V}, \mathcal{R}, \mathcal{U}\}, \tag{2}$$

where $C = \{1 \dots C\}$ is the set of agent classes, We denote with MA^c an agent of class $c \in C$; $\mathcal{M} = \{1 \dots M\}$ is the set of message types; $\mathcal{V}$ is the finite space over which Markovian Agents are spread; $\mathcal{R} = \{\rho^1(\cdot) \dots \rho^C(\cdot)\}$ is a set of C agent density functions; $\mathcal{U} = \{u_1(\cdot) \dots u_M(\cdot)\}$ is a set of M perception functions.

The perception function: $u_m : \mathcal{V} \times C \times \mathbb{N} \times \mathcal{V} \times C \times \mathbb{N} \to \mathbb{R}^+$ is defined such that the values of $u_m(\mathbf{v}, c, i, \mathbf{v}', c', i')$ represent the probability that an agent of class c, in position $\mathbf{v}$, and in state i, perceives a message m generated by an agent of class c' in position $\mathbf{v}'$ in state i'.

In this work discrete space is considered because MAGNET does not implement continuous space in the actual version. This means that space locations could be identified with an index; we denote with V the number of discrete space areas in the model thus $\mathbf{v} \in \mathcal{V} = \{0, \cdots, V-1\}$. In these conditions, agent density function $\rho^c(\mathbf{v})$ returns the number of class c agents in position $\mathbf{v}$.

2.1 M^3AM analysis

In this Section, we remind analytical method used to analyze a M^3AM model, particularizing it to the specific case of discrete space.

Let us denote $\rho_i^c(t, \mathbf{v})$ the number of agents in state i in position $\mathbf{v}$ at time t. We collect them into a vector $\boldsymbol{\rho}^c(t, \mathbf{v}) = [\rho_i^c(t, \mathbf{v})]$. Even if the total number of class c agents $\rho^c(\mathbf{v})$ remains constant over time, it dynamically varies its distribution over the set of states of the agents, meaning that:

$$\sum_{i=1}^{n_c} \rho_i^c(t, \mathbf{v}) = \rho^c(\mathbf{v}) \,, \quad \forall t \geq 0, \forall \mathbf{v}, c. \tag{3}$$

The final goal of a M^3AM analysis is to compute the transient evolution of $\boldsymbol{\rho}^c(t, \mathbf{v})$. We start by defining $\beta_j^c(m)$ as the total rate at which messages of type m are generated by an agent of class c in state j:

$$\beta_j^c(m) = \lambda_j^c \, g_{jj}^c(m) + \sum_{k \neq j} q_{jk}^c \, g_{jk}^c(m). \tag{4}$$

In eq. (4), the first term in the sum gives the rate at which messages are emitted when the MA remains in the state j (λ_j), taking into account the probability $g_{jj}^c(m)$; similarly, the second term is introduced to accumulate the rates of messages generated during state transitions, considering the transition rate q_{jk}^c and the generation probability $g_{jk}^c(m)$.

The rate $\beta_j^c(m)$ can be used to compute $\gamma_{ii}^c(t, \mathbf{v}, m)$, the total rate of messages of type m received by an agent of class c, in state i, at position $\mathbf{v}$, at time t. Let us consider a position $\mathbf{v}'$. In that point of the space there are $\rho_j^{c'}(t, \mathbf{v}')$ class c' agents in the state j; all together, they send messages of type m at rate $\beta_j^{c'}(m)\rho_j^{c'}(t, \mathbf{v}')$. A class c agent in position $\mathbf{v}$ and in state i receives just a portion of messages generated from a class c' agents in state j located in $\mathbf{v}'$. The fraction of messages received is determined by the perception function $u()$. The total rate of received messages is then:

$$u_m(\mathbf{v}, c, i, \mathbf{v}', c', j)\beta_j^{c'}(m)\rho_j^{c'}(t, \mathbf{v}').$$

Summing all the contributions coming from all the states and all the agent classes, and integrating over the entire area $\mathcal{V}$, the total rate of received messages is obtained:

$$\gamma_{ii}^c(t, \mathbf{v}, m) = \sum_{\mathcal{V}} \sum_{c'=1}^{C} \sum_{j=1}^{n_{c'}} u_m(\mathbf{v}, c, i, \mathbf{v}', c', j)\beta_j^{c'}(m)\rho_j^{c'}(t, \mathbf{v}') \tag{5}$$

that are collected in a diagonal matrix $\mathbf{\Gamma}^c(t, \mathbf{v}, m) = \text{diag}(\gamma_{ii}^c(t, \mathbf{v}, m))$. This matrix can be used to compute $\mathbf{K}^c(t, \mathbf{v})$, the infinitesimal generator of a class c agent at position $\mathbf{v}$ at time t:

$$\mathbf{K}^c(t, \mathbf{v}) = \mathbf{Q}^c + \sum_m \mathbf{\Gamma}^c(t, \mathbf{v}, m) \left[\mathbf{A}^c(m) - \mathbf{I}\right]. \tag{6}$$

The first term in the r.h.s. is the local transition rate matrix and the second term contains the rates induced by the interactions.

The evolution of the entire model can be studied by solving $\forall \mathbf{v}, c$ the following differential equations:

$$\begin{cases} \boldsymbol{\rho}^c(0, \mathbf{v}) &= \rho^c(\mathbf{v})\boldsymbol{\pi}_0^c \\ \frac{d\boldsymbol{\rho}^c(t, \mathbf{v})}{dt} &= \boldsymbol{\rho}^c(t, \mathbf{v})\mathbf{K}^c(t, \mathbf{v}) \end{cases} \tag{7}$$

From the number of agents in each state, it is easy to compute the probability of a class c agent at time t in position $\mathbf{v}$ in state i as:

$$\pi_i^c(t, \mathbf{v}) = \frac{\rho_i^c(t, \mathbf{v})}{\rho^c(\mathbf{v})}. \tag{8}$$

We collect all the terms in a vector $\boldsymbol{\pi}^c(t, \mathbf{v}) = [\pi_i^c(t, \mathbf{v})]$. Note that the definition of eq. (8) together with eq. (3) ensures that $\sum_i \pi_i^c(t, \mathbf{v}) = 1, \forall t, \forall \mathbf{v}$.

3 MAGNET SOFTWARE LIBRARY ARCHITECTURE

MAGNET is a modular library. It works by combining four different blocks, each one with a very specific task and organized as depicted in Figure 1. The implemented numerical analysis is entirely based on matrix calculus, thus this is the core part of the library. On top of it modules implementing the analysis are built. `ANALYSIS` block implements the fixed-point algorithm used to solve eq.(7), while `OBJECT HANDLING` provides all the data structures required for model description and their handler functions. The upper layer is the `API` block, which hides almost entirely the complexity of the implementation details. In the following Sections, we give a more detailed description of all the blocks.

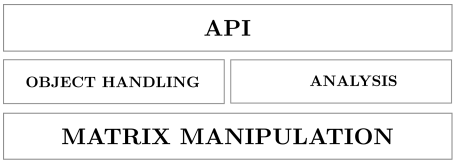

Figure 1: Graphical representation of MAGNET architecture

3.1 Matrix Manipulation

The core component is the one implementing all the matrix structures required by the library. Functions implementing matrix manipulation are not directly called during the model description, instead they are internally used by the library during the analysis. Structures provided by the matrix block are constituted by both dense and sparse matrices, direct and list-based vectors, along with all the required functions for their handling. By doing this, there is no need to directly call `libc` functions for memory management, being this part totally handled by the matrix functions. Creating a dense or a sparse matrix differs only in a flag passed as parameter to the matrices creation functions. It is worth to note that using dense or sparse matrices doesn't change functions to use, or their syntax. Once a matrix has been created (by providing to `CreateMatrixEnc()` a flag indicating the encoding, which will be memorized in the `Matrix` structure), users never need to care about the encoding. Along with matrices, we also introduced a `Vector` structure, which, in the simplest case, hides an array, but could store its data also in a list or a RB-Tree. In analogy with the matrix case, the encoding is decided during `Vector` instances creation. However, since the `Vector` sizes are known at compile time, and given that library doesn't perform data permutations during the analysis, the most efficient choice is to handle data as arrays.

3.2 Object Handling

A second major block is the one providing all the data structures required for handling models' objects. What we called `OBJECT HANDLING` is a layer implementing the discretization technique to solve eq. (7) introduced in Section 2.1. As in the theoretical model, the `n`-dimensional space is discretized into *cells*, with agents scattered all over this discrete space. MAGNET, once the space dimension is defined in the model, takes care of the correct identification of agents, identified by their names and coordinates. While functionally the library handle correctly the model, data structures are modeled in a different way with respect to the theory. As we recalled in the agents definition, an MA^c agent is defined by a tuple. A direct parallelism between formal and implemented structures has not been possible due to efficiency problems in algorithms. In MAGNET, there is a complex interconnection between data structures modeling agents, classes, and matrices. An agent is not implemented as a tuple, but as a structure keeping reference of its class, through an unique ID, and of the cell where it is located, identified by its coordinates. The central element is the class, which is the strongest link between agents and the matrices $\mathbf{Q}^c, \boldsymbol{\Lambda}^c, \mathbf{G}^c(m)$ and $\mathbf{A}^c(m)$, through internal identifiers. All this complexity is totally managed by the `OBJECT HANDLING` block, which provides also the functions used by the above layer (`API`) and the `ANALYSIS` block. With this architecture, neither the API nor the solution implementation require direct access to the data structures.

3.3 API

The API layer, built on top of the other three, is devoted to totally hide the underlying complexity, by providing a powerful set of functions which let write models in the most expressive way as possible. The functions of the `API` block follow some rules we strongly used during development process. An API function should be: (1) easy to read and understand, (2) with a very specific and simple task, (3) easy to use and compose with other functions of the API. This three requirements are the foundation of the API layer.

Almost all the functions work on the `M3AM` object, devoted to keep track of the whole model, and with labels identifying agents, messages and classes. The result, as we will show in Section 5, is a C source file describing the model which is easy to read, nearly as plain English. By achieving the three requirements presented, it results natural to describe a model, no matter how complex it is, without need of directly handling implementation details. This also helps in preventing some common C errors linked to manual de/allocation and pointers arithmetic.

As previously stated, the API layer provides functions making easy to describe simple or complex models. The steps required for the description of a model are: (1) creation of a new model, (2) definition of classes, (3) definition of agents, (4) creation of vectors $\boldsymbol{\pi}_0$ and $\boldsymbol{\Lambda}$, (5) creation of matrices $\mathbf{A}, \mathbf{G}, \mathbf{Q}$, (6) definition of messages, (7) definition of perception functions, (8) definition of analysis parameters, (9) analysis. MAGNET provides all the functions to implement the previous steps. In Section 5, we will show in more details the use of the API functions provided with MAGNET.

4 IMPLEMENTING MARKOVIAN AGENTS ANALYSIS

Once a model is described it can be analyzed by solving eq. (7). In MAGNET, we implemented conventional techniques discretizing time, and, in particular, we exploit the method in [7] in this first version of our library. For each agent, MAGNET numerically solves

the equations by implementing a fixed-point algorithm. Looking at the equation, we found out that, although every agent is linked to the others, the solution process could be split and executed in parallel. This is due to the fact that differential equation system has the same shape for all the *MA*s in the model and they are independent when $\mathbf{K}^c(t, \mathbf{v})$ is known. Since $\mathbf{K}^c(t, \mathbf{v})$ depends on all the *MA*s in the model, we implemented a multi-threaded fixed-point iteration algorithm, where every thread analyses a small number of agents and communicates with the other threads at each step.

4.1 Solution technique

`ANALYSIS` block implements transient analysis, that is assumed to be evaluated in the time interval $[0, T_{Max}]$. The time is discretized with a uniform step Δt, yielding $T_\Delta = \lceil T_{Max}/\Delta t \rceil$ discrete time points: $t \in \{0, \Delta t, \ldots, T_\Delta \Delta t\}$. The solution is then computed using an implicit method [7]. In particular, eq. (7) is approximated with:

$$\frac{\boldsymbol{\rho}^c(t + \Delta t, \mathbf{v}) - \boldsymbol{\rho}^c(t, \mathbf{v})}{\Delta t} \approx \boldsymbol{\rho}^c(t + \Delta t, \mathbf{v}) \mathbf{K}^c(t + \Delta t, \mathbf{v}). \quad (9)$$

Multiplying both the terms by Δt and reordering them, it is easy to obtain:

$$\boldsymbol{\rho}^c(t + \Delta t, \mathbf{v}) \left[\mathbf{I} - \mathbf{K}^c(t + \Delta t, \mathbf{v}) \Delta t\right] = \boldsymbol{\rho}^c(t, \mathbf{v}). \quad (10)$$

Assuming the initial condition in eq. (7) at $t = 0$, the solution vector $\boldsymbol{\rho}^c(t + \Delta t, \mathbf{v})$ is computed from eq. (10) starting from the knowledge of $\boldsymbol{\rho}^c(0, \mathbf{v})$. The method is implicit because matrix $\mathbf{K}^c(t + \Delta t, \mathbf{v})$ depends on $\boldsymbol{\rho}^c(t + \Delta t, \mathbf{v})$ itself thus eq. (10) is solved at every time step applying a fixed-point iteration algorithm. The number of required iterations is however very limited since $\boldsymbol{\rho}^c(t + \Delta t, \mathbf{v}) \approx \boldsymbol{\rho}^c(t, \mathbf{v})$.

Equation (9) is solved for every space point $\mathbf{v}$ and every *MA*s in it. This is why it is possible to split computation in different concurrent threads. In fact, assuming to know $\boldsymbol{\rho}^c(t + \Delta t, \mathbf{v})$, $\forall \mathbf{v} \in \mathcal{V}, \forall c \in C$, $\mathbf{K}^{c'}(t + \Delta t, \mathbf{v}')$ is computed only for the c' class *MA*s in position $\mathbf{v}'$ during fixed-point iterations, sharing the obtained temporary values of $\boldsymbol{\rho}^{c'}(t + \Delta t, \mathbf{v}')$ with all the other threads.

Storage complexity is limited thanks to the iterative numerical technique that allows to compute all the needed quantities at each iteration step. The size of the final computed vector $\boldsymbol{\rho}^c(t, \mathbf{v})$ at each iteration step is $O(|\mathcal{V}| \cdot N)$, where N is the greatest *MA* size.

4.2 Implementation details

A critical issue in multi-threading algorithm is the synchronization of threads on a temporal basis, since the results are correct if every thread is at the same temporal step t. Our solution was to delegate this task to an external function, which ensures that any thread continue its analysis until all the others are at its same point. By using semaphores, the fastest threads simply wait for the others to complete each iteration step. Besides, every thread signals when it finishes its local analysis. Synchronization process is detailed in the Algorithm 1.

Once the API function `AnalyzeModel()` has initialized all the structures required for multi-threading, computations of $\boldsymbol{\rho}^{c'}(t + \Delta t, \mathbf{v}')$ start, as described in Algorithm 2. To allocate local thread workload in terms of *MA*s, we used functions `Pos()` e `Class()` in Algorithm 2 that respectively return the position in the space $\mathcal{V}$ and the class of a given *MA* passed as parameter. As can be noted each thread waits for the signal (row 6 in Algorithm 2) from `AnalyzeModel()` (row 5 in Algorithm 1) and starts to compute $\boldsymbol{\rho}_i^{c'}(t + \Delta t, \mathbf{v}')$ at the time step only when all the threads have a stable value. This is ensured by `AnalyzeModel()` because it does not go on a new time step untill all the threads do not complete their computation (rows 6-7 in Algorithm 1).

Algorithm 1: Thread synchronization function

```
1  begin
2      t = t_0 + Δt
3      while t < t_max do
4          for Every thread do
5              Post on threads_sem
6          for Every thread do
7              Wait on sync_sem
8          t = t + Δt
```

Algorithm 2: Thread computation

input : Maximum iterations number i_{max}, time step Δt, maximum time t_{max}, tolerance ϵ, set of agent id to analyze A
local : i for iterations count initialized to 0, t, d for tolerance checking

```
1  begin
2      t = t_0 + Δt
3      for c ∈ C do
4          Generate I^c matrix
5      while true do
6          Wait on threads_sem
7          if t ≥ t_max then
8              Exit
9          for a ∈ A do
10             v' = Pos( a )
11             c' = Class( a ) do
12                 if i = 0 then
13                     for ∀m ∈ M, v ∈ V do
14                         Collect γ^{c'}(t − Δt, v', m)
15                 else
16                     for messages ∈ M do
17                         Collect γ^{c'}(t, v', m)
18                 Compute K^{c'}(t + Δt, v') matrix
19                 ρ_i^{c'}(t + Δt, v') =
                       ρ^{c'}(t, v') [I − K^{c'}(t + Δt, v')Δt]^{-1}
20                 d = ‖ρ_i^{c'}(t + Δt, v') − ρ_{i−1}^{c'}(t + Δt, v')‖
21             while (i < i_max) ∧ (d > ε)
22             Save results in agent file
23         t = t + Δt
24         Post on sync_sem
```

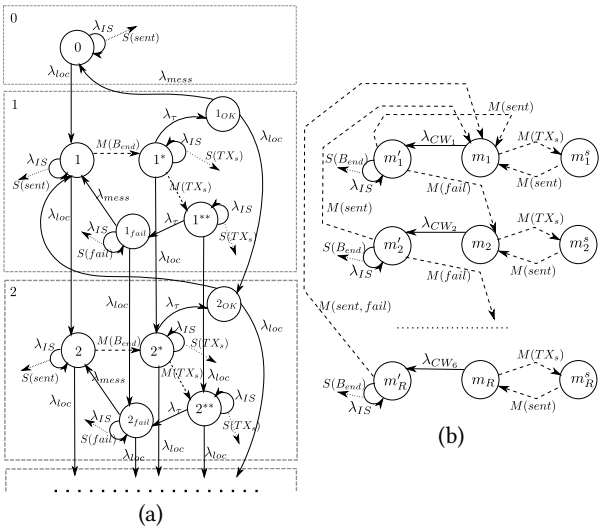

Figure 2: The *Buffer* (a) and *Backoff* (b) *MA* classes

Both the algorithms presented above wait and post on semaphores, through the objects we called `thread_sem` and `sync_sem`. With those terms, we refer to two arrays of semaphores, with length equal to number of threads. We need both arrays to manage the two synchronization functions. In more detail, each element of `thread_sem` is a semaphore used by the synchronization function to send signals to agents' threads. Similarly, `sync_sem` is an array of semaphores (again, one per thread), used by agents' threads to send signals to the synchronization function. Synchronization function shares the t parameter with the other threads, and it solely can change its value. With this technique, MAGNET ensures that all threads are stopped and started in the most efficient way as possible.

5 AN EXAMPLE

As reference example, we used the model presented in [9]. It describes two interacting wireless nodes operating with the 802.11 protocol. The only difference introduced here is that we considered three nodes and assumed all of them can directly exchange frames with the others. Here we do not describe how the model works, being out of the scope of this paper, but the interested reader can found deep details in [9]. Each wireless station in the *M3AM* model is made of the two *MA* classes, *Buffer* and *Backoff*, depicted in Figure 2. Thus the complete model is made of six agents located in three positions of a bi-dimensional space. Seven different types of messages are used and exchanged both between two *MA*s in the same position and among all the *MA*s spread over the entire space.

First of all, a model instance has to be created in MAGNET; we named the reference example `WN` and initialized it with

```
M3AM *model = SetModel(``WN'', DIM_VSPACE);
```

that gives a reference (in the variable `model`) to an `M3AM` object to use in the other functions. The *Buffer* MA class is created with

```
SetClass(model, ``Buffer'', nBuffer);
SetPI(model, ``Buffer'', pibuffer);
SetLambda(model, ``Buffer'', labuffer);
```

Variable	Value	Description
`DIM_VSPACE`	2	Space dimension
	$c = Buffer$	
`nBuffer`	71	Number of *Buffer MA* states
`pibuffer`	-	π^c of *Buffer MA*
`labuffer`	-	*Buffer* selfloop rate vector λ^c
`Qbuffer`	-	Local transition rate matrix $\mathbf{Q}^c$
	$c = Backoff$	
`nBackoff`	21	Number of *Backoff MA* states
`piBackoff`	-	π^c of *Backoff MA*
`labackoff`	-	*Backoff* selfloop rate vector λ^c
`Qbackof`	-	Local transition rate matrix $\mathbf{Q}^c$

Table 2: Parameters used in the example *M3AM* model

```
SetQ(model, ``Buffer'', Qbuffer);
```

We reported in Table 2 the variables used together with their types and value, when possible. Vectors and matrices used as argument are filled by using the primitives made available in MAGNET; as an example `pibuffer`, i.e. the $\boldsymbol{\pi}_0$ for the class *Buffer*, is instantiated with `NewPiVec()` and filled with `WritePiVal()` before linking it to the belonging class. Similarly, class agent *Backoff* is created using parameters in Table 2.

After classes definitions, we defined the messages and chained each of them with their own generation and perception probability matrices, $\mathbf{G}^c(m)$ and $\mathbf{A}^c(m)$ respectively. These definitions are simply done in MAGNET; for example, message *sent* definition into class *Buffer* is implemented as follow

```
SetMSG(model, "sent");
SetG(model, "Buffer", "sent", Gsbuffer);
SetA(model, "Buffer", "sent", Asbuffer);
```

where `Gsbuffer` and `Asbuffer` are the matrices created and filled with the primitives `NewG(nBuffer)`, `WriteGVal()`, `NewA(nBuffer)` and `WriteAVal()`.

Once classes and the messages are created it is possible to deploy *MA* instances associating to the appropriate space positions a given number of *MA*s identified with their names (a simple string); in the reference example, we put a *Buffer MA* in position `coo` and named it *Buff0* invoking the function `SetAgent(model, ``Buff0'', ``Buffer'', coo, 1.0)`. The name set during *MA* deployment is used by MAGNET to create a CSV file per *MA* storing transient state probabilities, if the analysis is started in *write* mode; in this case the file is named with the *MA* name (e.g. "`Buff0`" and `.csv` as file extension.

The last, and most cumbersome, thing to do to finalize `WN` is to define the perception function $u_m()$, $\forall m \in \mathcal{M}$. To setup perception functions is cumbersome because a value for each couple of states of two *MA*s anywhere located in the space is needed. MAGNET implements messages iterator (through the primitive `MessageIterator()`) to scan all defined messages in a model, and the `MessageIs()` and `isValid()` functions to check messages types. Function `SetPerception()` is used to set a matrix to a couple of *MA*s deployed in two positions. Thus the typical C code to use is:

```
Matrix *tempPercMat;
Message message;
message = MessageIterator(model);
```

	Intel		AMD	
# **Thread** (i)	T_i	s_i	T_i	s_i
1	189.97	0%	415.20	0%
2	133.00	30%	271.89	35%
3	123.46	35%	228.70	45%
4	141.77	25%	265.00	36%
5	139.17	27%	266.22	36%
6	151.37	20%	283.80	31%

Table 3: Parameters used in the example *M3AM* model

```
while (isValid(message))
  if (MessageIs(message, "sent")) {
        .....
     Fill in tempPercMat
        .....
     SetPerception(model, message, "Backoff", "Buffer",
                   coo1, coo2, tempPercMat);
        .....
  }
  if else ((MessageIs(message, "....")) {
        .....
  }
```

In the reference example, we needed to define seven `if-else` blocks, since seven message types are used in the model.

It is worth to note that the total number of states used in the model was 276 instead of more than 3 billions needed to implement the same model with an all-embracing CTMC.

We measured the performance of MAGNET by solving `WN` model in two different test-beds: a blade server equipped with a Dual Core AMD Opteron(tm) Processor and 4 GBytes of central memory (identified as `AMD` in the following), and a PC equipped with an Intel(R) Core(TM) i7-5500U CPU @2.40GHz and 8 GBytes of central memory (identified as `Intel` in the following). Transient analysis from $t = 0$ *sec*. to $t = 10.0$ *sec*. has been performed changing the number of used threads. A set of 40 runs has been executed for each thread configuration on both the two test-beds and the execution times have been collected. 90% of confidence intervals were very narrow thus we considered only the mean value as estimation of execution time. Obtained results are summarized in Table 3 and depicted as graph in Figure 3.

Let T_i be the estimated completion time when solution has been computed by using i threads; we evaluated the percentage speed up when i threads have been used as

$$s_i = \frac{T_1 - T_i}{T_1} \cdot 100 \tag{11}$$

Performance measures show that the use of multi-threading implemented in MAGNET is effective; in fact, in both the test-beds execution time decreases when more than one thread is used. This is due to the fact that both the processors are dual core CPUs, with the Intel CPU supporting *hyperthreading* technology, thus real parallelism supports the execution. The best speedup is obtained when the analysis is done with three threads because in this conditions threads have the same workload made of two *MA*s belonging to both the classes *Buffer* and *Backoff*. In all the other conditions, performance anyway improve due to the parallel execution but less effectiveness is exhibited because threads work with unbalanced workloads.

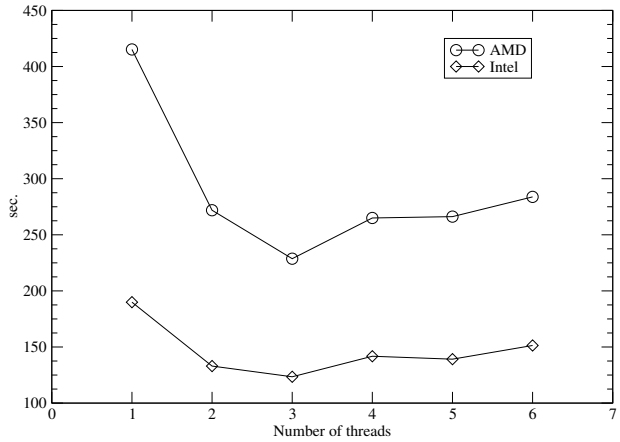

Figure 3: Computing time to solve the example model vs thread number

6 CONCLUSIONS AND FUTURE WORK

We described MAGNET, a software library to analyze Markovian Agents models. Two its major features are the parallel calculus in terms of number of agents, and the high-level API able to describe complex models with little effort. We showed the efficiency of the implemented multi-threading algorithm. As future work, we will extend MAGNET with a modern error handler and a memory management system based on memory pools. We also want to extend functionalities of MAGNET by implementing steady state analysis and a smart thread workload balance algorithm. MAGNET library is available at http://perf.unime.it/magnet.

REFERENCES

[1] A. Bobbio, D. Cerotti, and M. Gribaudo. 2009. Presenting Dynamic Markovian Agents with a Road Tunnel Application. In *MASCOTS09*. IEEE-CS.
[2] D. Bruneo, M. Scarpa, A. Bobbio, D. Cerotti, and M. Gribaudo. 2009. Analytical modeling of swarm intelligence in wireless sensor networks through Markovian Agents. In *VALUETOOLS09*. ICST/ACM.
[3] D. Cerotti, M. Gribaudo, and A. Bobbio. 2008. Analysis of on-off policies in sensor network using Markovian agents. In *4-th Int. Workshop PerSens*. 300–305.
[4] D. Cerotti, M. Gribaudo, and A. Bobbio. 2008. Disaster Propagation in Heterogeneous Media via Markovian Agents. In *3rd International Workshop on Critical Information Infrastructures Security*.
[5] G. Ciardo, J. Muppala, and K. Trivedi. 1989. SPNP: stochastic Petri net package. In *Proceedings of the Third International Workshop on Petri Nets and Performance Models, PNPM89*. 142–151. https://doi.org/10.1109/PNPM.1989.68548
[6] Vashti Galpin. 2008. Towards a spatial stochastic process algebra. In *7th Workshop on Process Algebra and Stochastically Timed Activities (PASTA)*. Edinburgh, UK.
[7] W.L. Miranker. 1981. *Numerical Methods for Stiff Equations*. Reidel, Dordrecht.
[8] R. Sahner, K. S. Trivedi, and A. Puliafito. 1995. *Performance and reliability analysis of computer systems: an example based approach using the SHARPE software package*. Kluwer Academic Publishers, Boston.
[9] Marco Scarpa and Salvatore Serrano. 2017. *A New Modelling Approach to Represent the DCF Mechanism of the CSMA/CA Protocol*. Springer International Publishing, Cham, 181–195. https://doi.org/10.1007/978-3-319-61428-1_13
[10] Anton Stefanek, Maria G. Vigliotti, and Jeremy T. Bradley. 2009. Spatial extension of stochastic π calculus. 109–117.
[11] Kishor S. Trivedi. 2002. *Probability and Statistics with Reliability, Queuing and Computer Science Applications* (2nd edition ed.). John Wiley and Sons Ltd., Chichester, UK.