A Multi-Robot GSPN Software Framework to Execute and Visualize Plans

Pedro Alegre Caldeira
Instituto Superior Técnico.
Lisbon, Portugal
pedro.a.caldeira@tecnico.ulisboa.pt

***Abstract*—Currently one lacks a tool that integrates modelling, visualization, logic and performance analysis and execution. Generalized stochastic Petri nets (GSPNs) are a mathematical model that have proven to be efficient for modelling homogeneous multi-robot systems due to their compact form, asynchronous execution and ability to capture temporal uncertainty. And so, our work extended an existing multi-robot modelling and task analysis tool that uses GSPNs.**

The main goal of this master's thesis is to develop a software package that allows the execution of GSPN plans in multi-robot systems and in multi-agent systems. In order to do so, we created two new modules: an execution and a visualization module. The execution module is responsible for executing the input GSPN while the visualization module's goal is to simulate or visualize the execution of the GSPN, depending on the intentions of the user. The integration with robots is assured by the robot operating system middleware (ROS).

In order to test our framework, we ran a series of tests on the part of the framework that was integrated with ROS. On the first set of tests, we only ran our framework and on the second, we used our framework alongside with the Gazebo simulator.

The obtained results for the tests where we only ran our framework show that, for a system with only one robot, it is possible to execute and visualize considerably large networks. On the other hand, when we used the simulator, we verified that it is possible to successfully run multi-robot systems with up to three virtually simulated robots, however, our tool did not support the computational overhead involved in the simulation of a team of four robots or more.

***Index Terms*—Generalized stochastic Petri nets, Multi-robot systems, Plan execution, Plan visualization**

I. Introduction

It has long been recognized that there are several tasks that can be performed more efficiently and robustly using multiple robots. Applications such as: inspection [1], surveillance, search and rescue [2], mapping of unknown or partially known environments [3] or transportation of large objects greatly benefit from the use of multi-robot systems.

Presently, an issue of this area is the fact that the solutions for these applications tend to be hand crafted almost every time a new problem occurs and on most cases, they don't assure any formal guarantees. Besides this, if the problem in hand is too complex, the designer will have difficulties coming up with an adequate solution, which leads to inefficient and unreliable results.

A good way to solve the mentioned issues is by using a formal model, such as generalized stochastic Petri nets (GSPNs), because it provides methods to synthesize policies to coordinate the multi-robot system that respect formal requirements. Besides this, they also present great advantages in the modelling of homogeneous robotic systems with its intuitive analysis of flow of information and control.

In most cases, the process of building a GSPN starts out by iteratively building a model and analyzing it until the obtained one formally guarantees the existence of certain properties. Next, a policy is obtained by an optimization method and finally, the plan is executed, taking into account the acquired policy and the built network.

Nowadays, one lacks a tool that unifies the above mentioned process [4], and consequently, the user has to implement interfaces that connect distinct 3rd-party tools, many of which implemented in different programming languages, which obviously is a cumbersome task.

The main objective of this master thesis, is to improve on an already implemented GSPN software framework developed in [5], in order to allow the visual representation of the model and the execution of GSPN plans. The main two components that were added were the execution module and the visualization module. The former allows the user to execute a GSPN plan, which represents a set of actions that an agent or a robot must perform. To execute a GSPN is to complete the mentioned actions. The latter is a graphical front-end, that provides an user friendly interface for the visualization of the GSPN model, the execution progress and the properties obtained from the model analysis. On Figure 1 we included the framework's final architecture where the arrows represent data flow. The yellow components are the modules that were introduced. The execution module uses the GSPN created with the GSPN module and sends the changes of the marking of the GSPN into the visualization module, which are posteriorly reflected on the visualization module's interface. The visualization module also uses the tools module to parse the GSPN into an object that it can understand.

Our framework is integrated with the robot operating system (ROS) middleware which enables the execution of plans in robots, but we also built a standalone version which does not need ROS to be used. This standalone version will not be discussed in this report. On Figure 2, we included a detailed version of the architecture of the two modules that we introduced. The Figure is divided in half to explicitly show the two different implementations. On the left side, we have the ROS integrated version, where the ROS execution module communicates with the ROS online visualization module. As portrayed by the Figure, the ROS integrated version does not have an offline visualization. On the right side, we have the standalone version, where the standalone execution module communicates with the standalone online visualization module. On the other hand, the offline standalone visualization module is not connected with the execution module because it is used to simulate a GSPN, instead of executing it.

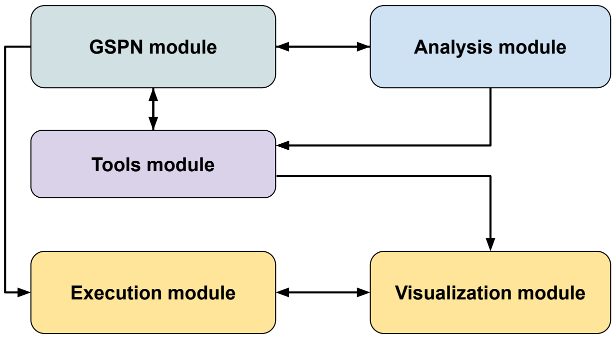

Fig. 1. Framework final architecture.

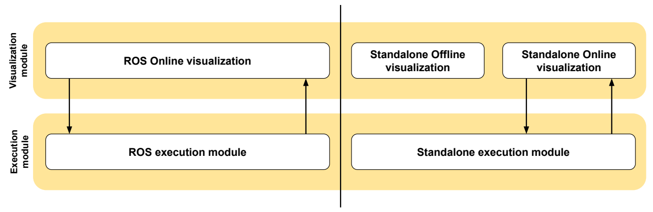
Visualization module
Execution module
ROS Online visualization
Standalone Offline visualization
Standalone Online visualization
ROS execution module
Standalone execution module

Fig. 2. Visualization and execution module.

II. Related Work

Many ways of modelling multi-robot systems have been proposed but as of today, the main ones are finite state automata (FSA), belief-desire-intention systems (BDI) and Petri nets (PN).

A. Finite state automata

A finite state automata (FSA) is composed by nodes, which represent states, and arcs, which represent transitions between the nodes (Figure 3). FSA based approaches have been very successful in modelling robotic systems and tasks because of their intuitive approach to design: FSAs only have two building blocks and they are relatively simple to build.

When comparing a FSA to a GSPN, GSPNs are usually smaller and although the growth of the size of the marking process is exponential for both models, in the case of GSPNs, it is possible to model, in a finite way, FSAs that are theoretically infinite. Although the construction of a FSA can be very intuitive and simple, most systems built with this approach tend to represent single-robot systems, since the existing mechanisms to build concurrency are not very intuitive and less explicit than on GSPNs.

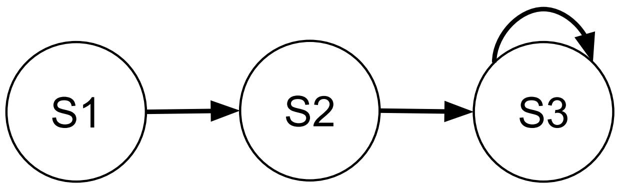

Fig. 3. FSA example

B. Belief-desire-intention

Belief-desire-intention systems, or BDI, appeared as an alternative to FSA-based approaches. To be simply put, BDI [6] is an architecture where the agent has three main components:

- Beliefs, which represent what the agent knows about the environment;
- Desires, which represent the agent's goals and as such, some can represent a possible end state;
- Intentions, which represent the selected behaviours to be executed.

The main reason why this approach is an evolution in comparison with the previous one is that with BDI, the programmer is not forced to create a very solid list of tasks to be completed. Instead, the robot itself has a certain degree of freedom to choose what he feels is more adequate to execute at the time.

C. Petri nets

There are many tools with the purpose of modelling, visualizing and analyzing PNs. However, the main limitation of these tools is the fact that they don't do all of these tasks in an integrated way, causing the programmer's job to be more difficult than it has to be.

PRISM [7] is an open-source probabilistic model-checker. It provides mechanisms to build and analyse DTMCs, CTMCs, MDPs and the extensions of these models with rewards. Other DTMC and CTMC analysis tools are available but unlike PRISM, they do not allow logic specifications. However, being focused on model-checking, PRISM does not allow execution of plans.

STORM [8] is another probabilistic model checker like PRISM, but since it was developed more recently, it's more optimized than the former. However, STORM doesn't support support PRISM features such as probabilistic timed automata and multi-objective model checking.

SMACH [9] is a ROS-integrated Python framework used for modelling of robotic systems based on FSA. It was created with simplicity in mind so that any programmer who needs a small state machine to define the behavior of its robot can build it rapidly and intuitively. Unlike usual FSAs, SMACH allows parallel execution, using the execution policy *Concurrence*. Nevertheless, this concurrency is not very explicit. Moreover, it doesn't show the passing of time, which can be an essential metric do analyze in a multi-robot system. Adding to this, this tool doesn't allow formal analysis.

Pipe [1] is a tool to design GSPNs and PNs. It has as simple user interface and moreover, you can also simulate the token game, which means that we can observe the tokens being exchanged between the places. However, it doesn't allow an execution of the network.

GreatSPN [10] is another tool created with the purpose of analysing Discrete Event Dynamic Systems (DEDS), or more concretely, GSPNs. As of today, the tool allows the user to build the GSPN, visualize it in a graphical way, evaluate the network's qualitative and quantitative properties and finally, visualize the obtained results. Although this tool allows the user to model and analyse a certain GSPN, it doesn't allow its execution.

Contrary to the previously mentioned tools, Petri net plans [11] can execute a PN. However, they are based on PNs, which we can be considered as a slightly less expressive model when compared to GSPNs since they don't take into account uncertainty. Besides this, Petri net plans aren't very flexible in the sense that the user has to use a series of predefined building blocks that compose the framework and although these building blocks are a good way of creating a robust model, this limits the user's possibilities.

On Table I, we summarize the purpose of each tool and its main characteristics.

TABLE I
SOFTWARE TOOLS

Tool name	***Purpose***	***Modelling***	***Analysis***	***Execution***
PRISM	Yes	Yes	Yes	No
STORM	Yes	Yes	Yes	No
SMACH	Yes	Yes	No	No
PIPE	Yes	Yes	Yes	No
GreatSPN	Yes	Yes	Yes	No
PN Plans	Yes	Yes	Yes	Yes
Our framework	Yes	Yes	Yes	Yes

III. BACKGROUND

A. Generalized stochastic Petri nets

Definition 1. *Formally, a GSPN can be defined by the following tuple:*

$$GSPN = (P, T_I, T_E, F, W^-, W^+, m_0, Z_I, R) \qquad (1)$$

- *P is a finite set of places;*
- *T_I is the set of immediate transitions and T_E is the set of exponential transitions where $T = T_I \cup T_E$. Immediate transitions model activities that can occur in the system and fire as soon as they have the necessary number of tokens. Exponential transitions are characterized by an exponential distribution which models the elapsed time until firing;*
- *F is the set of arcs where $F = (P \times T) \cup (T \times P)$;*
- *$W^- : P \times T \to \mathbb{N}$ and $W^+ : T \times P \to \mathbb{N}$ are input and output arc weight functions, respectively. Input arcs go from places into transitions and output arcs go from transitions into places;*
- *$m_0 : P \to \mathbb{N}$ is the initial marking, which can be represented by a vector with a size corresponding to the number of places;*
- *$Z_I : T_I \to [0, 1]$ is the weight of each immediate transition, which means that on the case of having two enabled transitions, this value will determine the probability of each transition being fired;*
- *$R : T_E \to \mathbb{R}_{\geq} 0$ is a function that associates each exponential transition with a rate.*

[1]http://pipe2.sourceforge.net

More intuitively, a GSPN is composed by a net structure and a marking. The net structure is a bipartite graph built with four elements: places, immediate transitions, exponential transitions and arcs. Between two places, we always have a transition and both these elements are connected by arcs. The places are represented by circles, and the transitions are represented by rectangles. Places can be either input or output, depending on their interaction with a transition. Considering $T = T_I \cup T_E$, $t \in T$ and $p \in P$, the set of input places of t can be defined as $IN = \{p|(p, t) \in F\}$ and the set of output places of t can be defined as $OUT = \{p|(t, p) \in F\}$. The net structure of a GSPN will remain constant all throughout the execution.

The marking, on the other hand, is the discrete number of tokens inside each place and is defined by $S : P \to \mathbb{N}$. The marking is visually represented by small dots inside each place. The only way a transition is fired is if the said transition is enabled. A transition is considered to be enabled if each input place IN is marked with at least W^- tokens. If the transition is enabled, then it can be fired, removing W^- tokens from the input places and adding W^+ tokens to the output places. The tokens' configuration will change all throughout the execution and they're what empowers this tool: with tokens we can observe the evolution of the model during the passing of time.

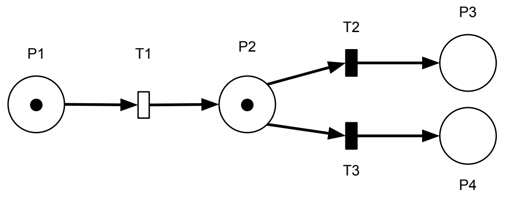

Fig. 4. A marked GSPN with an exponential transition (T1) and two immediate transitions (T2 and T3).

An important feature of GSPNs is the *marking graph* which formally can be defined as $< S, E >$ considering $E : S \times S \to T \cup \emptyset$, where T is a transition. We consider that E can be null in order to cover the case where we have two markings, that are not connected by a transition.

B. *GSPNs and multi-robot systems*

In our context, the tokens of a GPSN represent two different elements, depending on the places where they are: robots or counters. Tokens will be seen as robots when they are in a place where an action (such as moving into a specific room) can be performed (action places). Besides this, these kinds of tokens are always associated with a robot. On the other hand, tokens will be seen as counters when they are in a place where no action is being performed (resource places), meaning that their existence is merely informative to the designer and to the analysis and synthesis algorithms. These counters will be useful to count, for example, the number of robots that went through a certain place.

Immediate transitions model action selection while exponential transitions model uncontrollable events which can represent the reaction of the environment towards an action executed by the robot or an internal change.

As an example, consider the GSPN and the robot system illustrated on Figure 5. On the upper part, the reader can observe the GSPN with five places, four exponential transitions, two immediate transition and three tokens. One of the tokens is on P1, another one is on P3 and the third one is on P5. On the lower part, we have an illustration of the multi-robot system associated with this specific GSPN.

The GSPN represents a system where two robots must measure the temperature of a critical area. If the temperature is higher than a predefined threshold, an alarm will be activated. It is advantageous to use a multi-robot system because by doing so, we can have a constant monitorization on the critical system: when one of the robots is taking measurements, the other one is rebooting its system in order to avoid working for long periods of time. All places except P3 represent action places, where a specific action is being performed, while P3 represents a resource place, where no action is being performed. As such, both the tokens on P1 and P5 represent robots, while the token on P3 represents a counter. This token is not associated with any robot and on the case where the user checks the current marking, by checking the number of tokens inside P3, it will be clear that the number of measurements done is 1. Regarding the transitions, T4 and T5 are immediate transitions and represent the selection of an action, meaning that when a robot enters P2, it will decide its next step based on the current marking. The remaining transitions are exponential because we do not know how long it will take to accomplish any of the actions associated with them. For instance, when on P1, we do not know how long it will take a robot to check the temperature of the critical area, and as such, the associated transitions are exponential.

C. *Policies*

A policy defines how a certain system behaves at a given time. To be simply put, a policy dictates which actions should be taken in each state. A state in GSPNs is the current marking

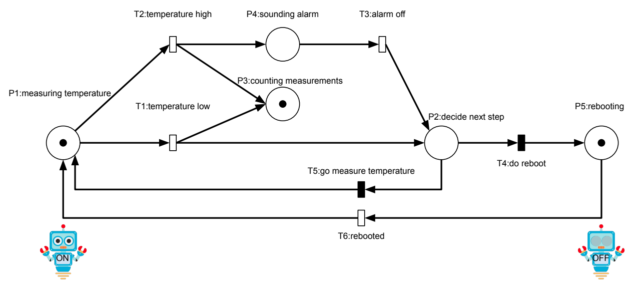

Fig. 5. A GSPN with the corresponding multi-robot system.

of our GSPN and an action is a subset of immediate transitions, t_i, where $t_i \in T_i$.

Definition 2. *A policy is formally defined as:* $\pi : ST \rightarrow A$*, where* ST *is a set of states and* A *is a set of actions. In the context of GSPNs, an action* A *can be defined as a set of transitions* T *and associated probabilities* Pr*,* (T, Pr)*. We can have four different forms of policies (which are interchangeable), such as deterministic, stochastic, stationary and non-stationary. However, we will only focus our attention on stationary and deterministic policies since this will be the kind that we will use further on. Deterministic are the simplest cases of policies and there are no uncertainties to which action the system will execute on each state. Stationary policies are policies that don't change over time.*

IV. Implementation

A. *The execution module*

This module was developed to be used with the ROS Noetic distribution and was developed and tested with Ubuntu 20. In a high level overview, each robot will be seen as a token and each action will be seen as a place of the GSPN. By using and executing plans of GSPNs, we are creating a generally transparent system where the user can easily understand what each robot is doing at every point in time because each place is directly associated with a specific action. If a robot is, for instance, stuck in place P1, then the programmer can have the intuition that the reason can be inside the code of the action.

Generally, the algorithm created to perform the execution takes as input a GSPN, a policy, a mapping between the actions and each place (for a GSPN with five places, we will have 5 actions available to be executed), a flag to determine the amount of output information and a list with the resource places. When a robot, which is represented by a token, enters a place, it will start executing the corresponding action. When it is done with it, the action outputs either the transition to fire in the GSPN or a flag that requires the algorithm to check the input policy in order to determine the next transition that should be fired. Afterwards, the transition is fired and the robot moves into the next place, starting this process once again. As outputs, the user receives the token game, the action's feedback and the output transitions (Figure 6).

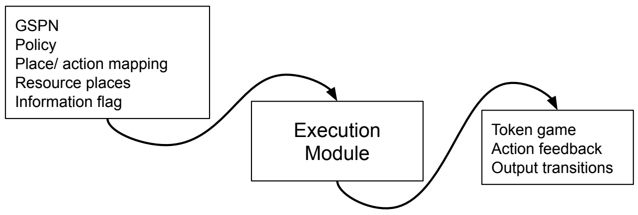

Fig. 6. ROS execution module inputs and outputs.

1) *General architecture:* In general, our system has a decentralized architecture because our framework's main goal is to be used with multi-robot systems and as such, it is not a wise choice to concentrate the entirety of our algorithm into a single point of failure. Each robot will have an execution node running inside it, which means that for each robot, we will have the same number of execution modules running.

2) *Inner robot communication:* The inner robot communication defines how each robot is able to execute the many actions of its plan. Our framework uses the actionlib [2], which is based on a client-server model where the action client communicates with an action server. Actions consist of three separate parts: a goal, which is firstly sent to the server, a result, which represents the output of the function that was executed and a feedback which can be provided to understand the system's progression.

Consider Figure 7, which represents a robot's inner structure. On the left side, we have the robot's action client and servers, while on the right side we have the current GSPN. For each place of the GSPN, we have one action server, which are named according to the places they represent. For instance, place P1 is associated with the P1 action server. Taking into account that the robot is on P1, then it will connect its action client to the P1 action server. This connection is represented by the bi-directional arrow between the action client and the P1 action server. In general, after completing the action, the server will either return a transition (if it is an exponential transition) or a flag that requires the policy and the current marking to be checked (if it is an immediate transition). Afterwards, the transition is fired, the robot will disconnect its action client from the current action server and connect it to the action server of the new current place. This process repeats itself until the robot's current place has no output arcs or until the user explicitly stops the execution.

Taking into account that every robot will have to run inside it one server per place of the GSPN, this means that for example, a GSPN with three places and three robots, a total of nine action servers will be created.

For a better understanding of the algorithm behind the execution, please consider Algorithm 1. The red and blue portions will be introduced later. For now, only consider the black colored portions of the pseudo-code. The algorithm takes as inputs a GSPN, the policy, the mapping between each place and server and a flag to decide whether the outter communication is performed via topic or service. The list of resource places is not used directly in this algorithm and as such, we chose to omit it. The first step is to know the current place of the robot, so that it can map it to the corresponding action server. Afterwards, the action is executed and once it is finished, the algorithm checks whether the robot's current place has output arcs or not. If not, then the execution is finished for the robot. On the other hand, if there are output arcs, then the algorithm checks the result provided by the action server. If the result equals to the string "None", this means that the transition to be fired is immediate. As such, the algorithm must take the current marking, check the policy that was provided and fire the resulting transition. On the other hand, if the result is a concrete transition, this means that the transition to be fired is exponential and as such, it is simply fired.

3) *Outer robot communication:* We chose to use ROS topics [3] to exchange information between robots. In its essence, a topic is a way of communicating between nodes through ROS using a publisher/subscriber architecture. When a message is published to a topic, every subscriber of the said topic executes a listener callback function. In our specific case, every robot is both a publisher and a subscriber to the exact same topic.

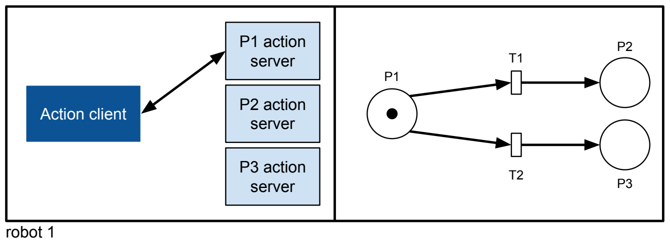

Fig. 7. ROS action client and servers and corresponding GSPN.

On Figure 8 you can analyze the outer communication model. In this example we have a multi-robot system composed by two robots and a GSPN with three places, which translates into three servers for each robot. Robot 1 is on P1 and as such, it connects its action client to the P1 action server. On the other hand, robot 2 is on P2 and as such, it connects its action client to the P2 action server. If one of the robots finishes the action, they will fire a transition and as such, the marking will change. These changes are published to the fired transitions topic and each subscriber of the same topic receives them. On Algorithm 1, the topic communication is performed by the red colored portions of the pseudo-code. Every time a robot fires a transition, it will have to publish to the topic the information that was previously mentioned, so that the remaining members of the team can update their local GSPNs.

Communicating through topics can, however, make the understanding of the network much harder since every time

[2]http://wiki.ros.org/actionlib

[3]http://wiki.ros.org/Topics

Algorithm 1: ROS execution algorithm.

```
Input: GSPN, policy, mapping between each place
       and server, communication flag to decide
       between topic and service
Output: Execution of actions
1  while True do
2      Get the robot's current place, connect to the
        corresponding action server and start executing
        action;
3      if The action server is done then
4          Check GSPN and get the output arcs of the
            current place;
5          if There are any output arcs then
6              Get the result returned by the action server;
7              if Result is immediate transition then
8                  Check the communication flag;
9                  if Communication flag is topic then
10                     Check transition to fire in the policy
                        and fire it;
11                     Share the firing information with
                        the topic;
12                 else if Communication flag is service
                    then
13                     Get the other robots' current places,
                        check the policy and get the other
                        robots' current states;
14                     if All robots are in state 'Done'
                        then
15                         Fire the transition obtained from
                            the policy;
16                         Share the firing information
                            with the topic;
17                     else
18                         Wait
19             else
20                 Fire the transition obtained from the
                    action server;
21                 Share the firing information with the
                    topic;
22         else
23             End execution of this robot.
24     else
25         Wait for result
```

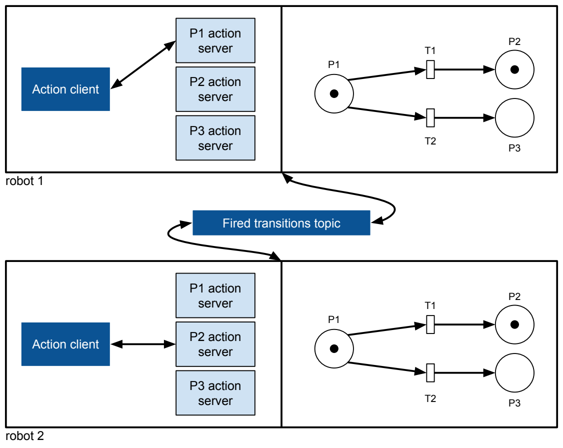

Fig. 8. Execution module outer communication.

there is a change, a message is published to the topic. We concluded that there are only two moments when the robots actually need to communicate their current place, which is the moment when the robot has to choose which immediate transition it will fire (left side of Figure 9) and the moment when the robot is on a synchronization case (right side of Figure 9). Taking this into account, we decided to create an extra communication channel, through ROS services [4], which instead of being called every time someone publishes to it, is only executed on these two moments.

To cover the first one, we defined a very simple service for each robot which returns its current place. Every time a robot needs to make a decision regarding which Immediate transition will be fired, it calls the service of every remaining member of the team and registers their current places. After having the quorum from every member, the robot builds the current marking of the GSPN and makes a decision regarding the transition to fire, based on the obtained marking.

Regarding the second moment, we used the previously introduced service and two new ones: a service to return the robot's current activity state and a service to change the robot's current place. Both these values are defined as global variables and the robot's current activity state is either *Done* or *Doing*, depending on whether it is done with its current task or not. Considering Algorithm 1, the blue colored lines represent, in a very simplified manner, the lines that were introduced to allow this communication mechanism's functionality. If the communication flag is set to service, then this means that the outer communication is performed via service. As such, the algorithm must get the other robots' current places, check the input policy and get the other robots' current states. If all robots are done, then the transition is fired, however, if some robots are still in the middle of the execution of an action, then it will wait for them to finish.

[4]http://wiki.ros.org/Services

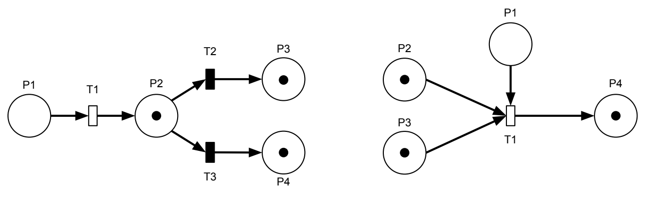

Fig. 9. Moments where robots need to communicate.

4) Resource tokens and resource places: Besides having robots on our GSPNs, we also have the possibility of having resource tokens and resource places:

Definition 3. *A resource place is a place of the GSPN where no action is executed and where instead of robots, we have resource tokens. These tokens can be used to take metrics from the execution of a GSPN.*

Considering Figure 10, **P2: Counting measurements done** is a resource place and as such, the tokens inside it do not represent any robot and therefore are not associated with the execution of any action. These tokens exist in this place in order to inform the user of how many measurements were done, through the analysis of the GSPN's marking. Every time transition **t1: Temperature measured** is fired, a new token is created both on **P2** and **P3**.

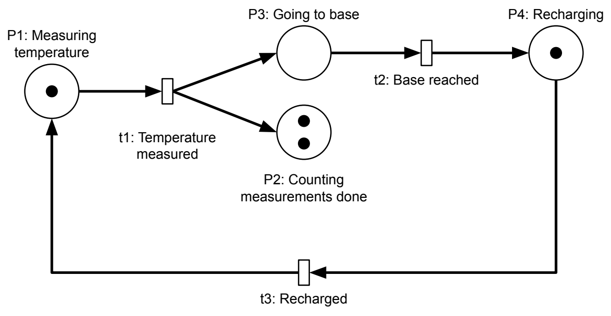

Fig. 10. Resources example.

5) Limiting the possible input GSPNs: Before the execution starts, our algorithm analyzes the GSPN that the user provided as input. This analysis depends on the list of resource places that the user also provides in the beginning of the execution. The algorithm creates a temporary GSPN, equal to the one provided and removes the resource places. Next, it obtains the reachability graph for this newly created GSPN and counts the number of tokens in each node of the graph. If the number of token sums changes in any part of it, then there is either creation or destruction of tokens, which cannot be allowed since we are working with robots, which are physical entities, and we reject the input GSPN. However, if the number stays constant throughout the reachability graph, we accept the GSPN.

B. The online visualization module

The visualization module's main goal is to visualize a GSPN and its execution.

The visualization module's architecture is composed by three different elements: a backend in Python 3; a frontend in HTML, CSS and Javascript that also uses Vis.js [5] to represent the GSPNs; and a web framework tool to allow the flow of information between the two previous elements, Flask [6]. The inner architecture of the visualization module is on Figure 11. The arrows represent data flow.

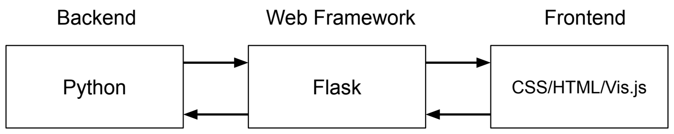

Fig. 11. Visualization module inner architecture.

By using ROS, we were able to take advantage of all its built-in functions and components. Topics and topic callbacks were crucial while building this module. Whenever a publisher of the topic mentioned in Section IV-A3 publishes to it, the backend of the visualization module registers this change and applies it to the GSPN presented in the frontend. This mechanism is illustrated on Figure 12 and in order to achieve it, we created a new subscriber to the above mentioned topic in the backend of the visualization module and associated a callback function to it, which saves the fired transition and the resulting marking into a Python list. On the frontend side, a periodic function that fetches the saved updates is executed every second and applies them.

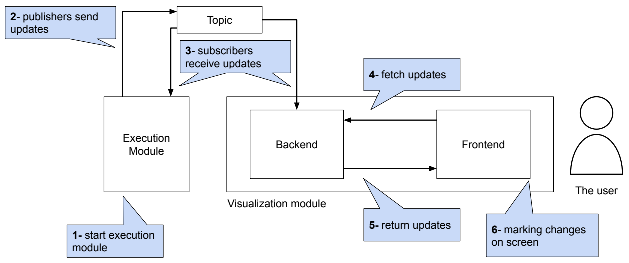
2- publishers send updates
Topic
3- subscribers receive updates
4- fetch updates
Execution Module
Backend
Frontend
The user
Visualization module
1- start execution module
5- return updates
6- marking changes on screen

Fig. 12. Interaction with ROS online visualization.

V. Results and discussion

We created two different types of tests: on the first batch of tests, we decided to test the scalability of our framework *without* the Gazebo [7] simulator. On the second batch of tests, we tested our framework with a multi-robot simulated system on Gazebo. The setup that was used consists of a laptop with an Intel Core i7-8550U CPU and 11 Gb of RAM.

[5]https://visjs.org/
[6]https://flask.palletsprojects.com/en/1.1.x/
[7]http://gazebosim.org/tutorials?tut=ros_overview

A. Testing the framework

We performed three tests without simulated robots. In all of them we measured the percentage of CPU and virtual memory (not including swap) that was consumed throughout the execution by using two functions from the python library, psutil [8], cpu_percent and virtual_memory respectively. For each moment of the test, we took a sample of five measurements and calculated the average of them. The baseline of each value was 0,8% of CPU and 25,9% of virtual memory

On the first test, we increased the number of robots and kept the number of places constant. The GSPN used is on Figure 13 and the results can be analyzed in Figure 15. Both values increase with an increasing number of tokens. Besides this, the time that the system took to start the execution also increased with the addition of more tokens, creating wait times of around 20 minutes, for more than 13 tokens.

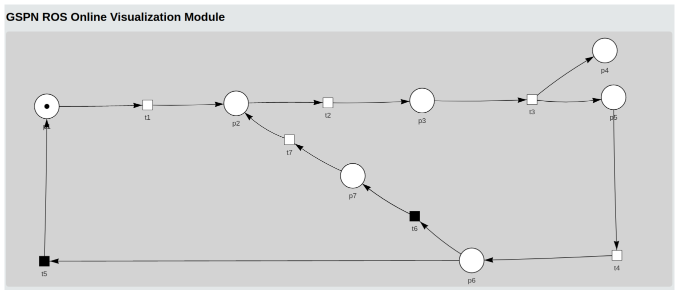

Fig. 13. GSPN to test the increase of tokens.

On the second test, we increased the number of places and kept the number of tokens constant. The initial GSPN is on Figure 14 and the results are presented in Figure 16. The percentage of CPU and virtual memory both increased with the increase of the number of places. Also in this test, as we inserted more places, the time it took the system to start the execution also took longer. And at times we had to wait around 20 minutes for the execution to begin.

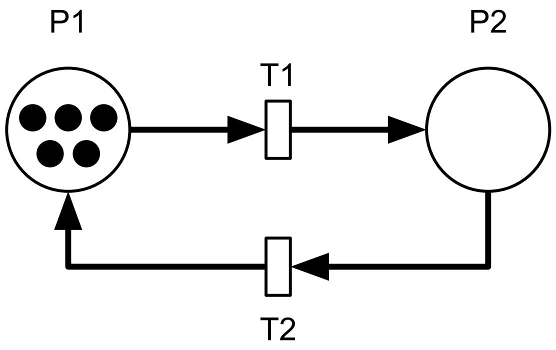

Fig. 14. Initial GSPN to test increase of places.

[8]https://pypi.org/project/psutil/

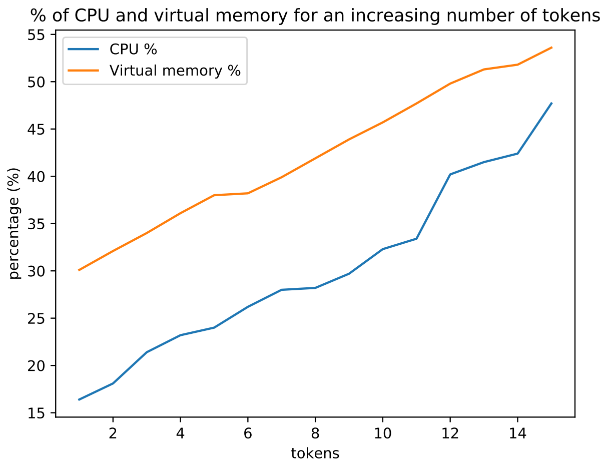

Fig. 15. Results for an increasing number of tokens.

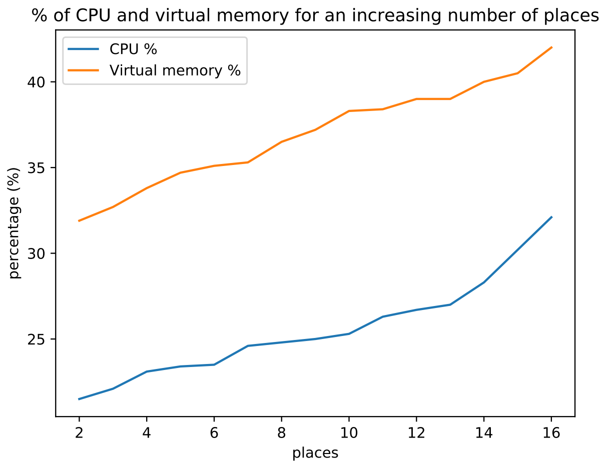

Fig. 16. Results for an increasing number of places.

The increase of both computational power and memory usage for these two tests is due to the fact that the more elements we introduce into our plan, the more resources will need to be provided by the host computer. The values for the virtual memory increased in a much more linear way, when compared to the increases of the processing power, because adding a new place or adding a new token (for example) usually takes almost the same amount of memory every time. When talking about the percentage of CPU, the values tend to reflect a bigger variation because as we add more tokens, more actions are being executed at the same time.

The wait time necessary to start the execution is related to the time that the processor requires to assign each task. Increasing the initial number of tokens means an increase on the number of action clients to match with action servers. Increasing the initial number of places, adds more action servers that can be paired up with action clients.

We took the percentage of CPU used from both these tests and related both values to the total number of action-servers and obtained the results on Figure 17. Taking into account the growth in action servers of each test, we can easily see that although on both cases the percentage of CPU increases heavily, on the first test, the percentage quickly surpasses the percentage of the second test. This occurs because on the first test, by adding one token, we are adding seven new action servers (one for each place), whereas on the second test, when we add one place, we are adding five new action servers (one for each token).

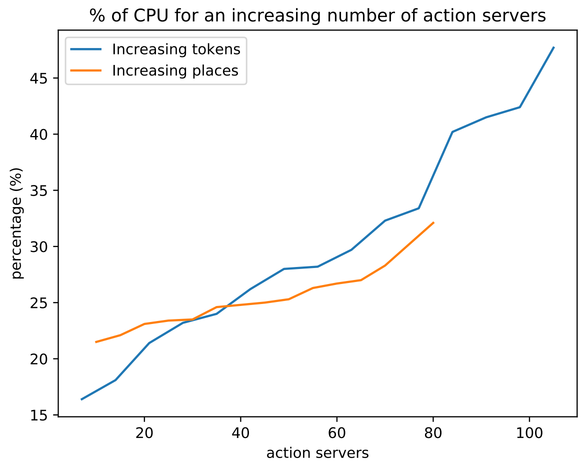

Fig. 17. Results for an increasing number of action servers.

On the final test, we increased the number of places and kept the number of tokens constant and equal to one. This test aims to understand whether our framework would possibly work well in a truly decentralized architecture or not. We started this test with two places, two transitions, but only one token, and went up to 35 places and afterwards we went up to 100, counting 5 by 5. The results are in Figure 18 and they are very similar to the ones obtained previously. However, we registered a major difference in terms of the time it took to initiate the execution. On the two previous tests, we pointed out that there was a major time gap between the moment we launched the command to start the execution and the moment the execution started. On this third test, the time gap was very close to zero.

The time gap registered on the previous two tests does not occur with test three because we only had one token to execute the plan, which means that the processor is only assigning one task at a time, even though it has increasingly more action servers active.

B. Testing the framework with simulated robots

Consider the turtlebot basic environment and suppose that it represents a room that needs 24/7 surveillance on the temperature of four different areas. A team of mobile robots

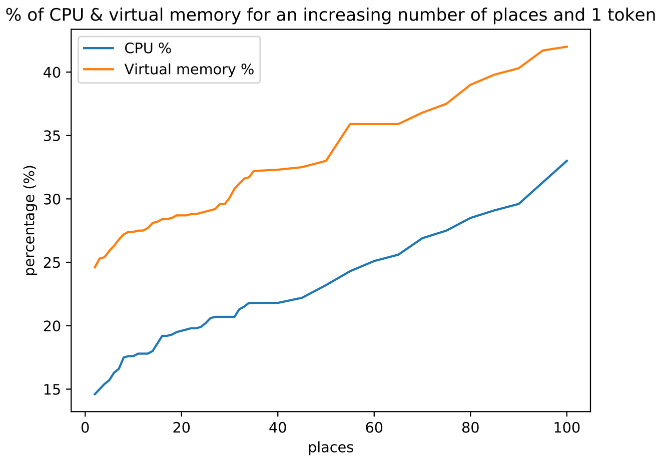

Fig. 18. Results for an increasing number of places with only one token.

is deployed in order to improve the efficiency of the monitorization task.

The goal of the team is to go around the room and perform a quick sweep of every single one of the four areas, meaning that depending on the size of the team, each robot will have to move more (if the team is composed by few members) or less (if the team is composed by many members).

We took the map of the turtlebot basic environment and divided it into four different locations: L1, L2, L3 and L4. With these four locations, we designed a topological map and obtained the one presented on Figure 19.

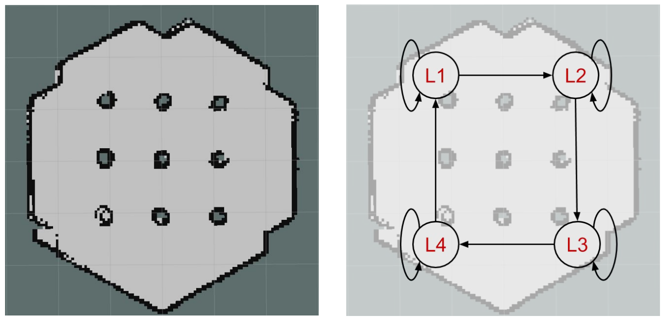

Fig. 19. Topological map with environment map.

The obtained GSPN is presented on Figure 20. The Figure is a screenshot taken from the ROS online visualization module.

We ran our setup with one, two, three and four robots. For the first three tests we managed to run an execution successfully, however, the test with four robots led to errors and was not successful. For each test, we filmed a short video and made it available through the following **link**.

When testing the system with one robot, the execution was both smooth and relatively fast.

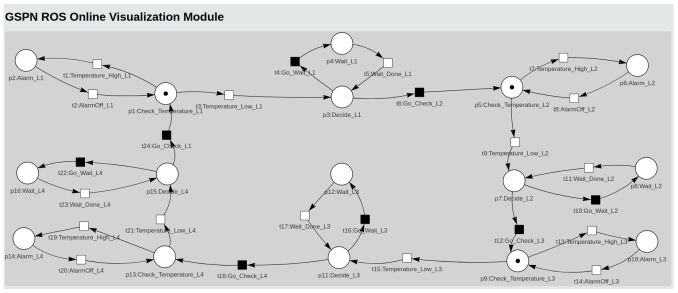

Fig. 20. Temperature patrol designed GSPN.

With an increasing complexity, our system started having some performance issues. When we tried running the test with two robots for the first time, we realized that instead of moving, the second robot would initiate its recovery behaviour, which is an embedded mechanism that attempts to clear the space around the robot. Although disappointing, we understood that we could bypass this issue by manually clearing the robot's costmaps. We had to do this in the beginning of the execution but after doing it a couple of times, the system was able to continue autonomously.

When running the test with three robots, we faced similar issues to the previous test, where only one of the robots managed to start moving autonomously. The remaining two got stuck in the beginning of the execution and we had to manually clear their costmaps. The movement of the simulated robots was generally not very smooth. Besides this, an important note to take is that on some cases, our framework would not apply the input policy when confronted with an immediate transition and the simulated robots would keep waiting for the corresponding action server indefinitely. Although we tried understanding the reason behind this issue, we were not successful due to the randomness of the event. However, we concluded that the system worked best when the places and the transitions represented on the visualization module were not moved around, hence the strangely organized left side of the video for the simulation with three robots.

Finally, we achieved the bottleneck of our tool on the test with four robots. On most cases, the robots would stop moving and not even the costmap clearing service would be able to recover them. On the presented video, the execution takes very long to start itself and some robots eventually get stuck in the environment and can never recover from that. When this happens, move base automatically discards the current objective and considers it done, which leads to moments like the one on timestamp 6:17, where we see robot 3 surpassing robot 4, which is not something that should happen due to the nature of our GSPN. This happens because robot 3 spent too much time on L1 without moving and at some point move base discarded that goal and with that, both robot 3 and robot 4 got entangled into the waiting area and eventually robot 3 was sent into L2 (while robot 4 stayed in the waiting area), moving in a horizontal manner and completely surpassing robot 4. In the end of the video, every robot was stuck and not moving, which of course led to the premature end of the execution of the plan.

VI. Conclusions

In this paper we introduced a multi-robot GSPN software framework to execute and visualize plans in systems with and without simulated robots.

Although the results were obtained from a simulated environment, taking into account that a vast majority of the framework's code was integrated with the ROS middleware and that when we tested a truly decentralized system, everything ran fairly well, we believe that, with more future work, this framework will work well in a multi-robot system with real robots.

To improve this framework, tests with real robots should be performed, so that it can be tested to its limit in a real life scenario.

References

[1] Kelin Jose and Dilip Kumar Pratihar. Task allocation and collision-free path planning of centralized multi-robots system for industrial plant inspection using heuristic methods. *Robotics and Autonomous Systems*, 80:34–42, 2016.

[2] Ricardo Mendonça, Mario Monteiro Marques, Francisco Marques, André Lourenco, Eduardo Pinto, Pedro Santana, Fernando Coito, Victor Lobo, and José Barata. A cooperative multi-robot team for the surveillance of shipwreck survivors at sea. In *OCEANS 2016 MTS/IEEE Monterey*, pages 1–6. IEEE, 2016.

[3] Luca Carlone, Miguel Kaouk Ng, Jingjing Du, Basilio Bona, and Marina Indri. Simultaneous localization and mapping using rao-blackwellized particle filters in multi robot systems. *Journal of Intelligent & Robotic Systems*, 63(2):283–307, 2011.

[4] Matt Luckcuck, Marie Farrell, Louise A Dennis, Clare Dixon, and Michael Fisher. Formal specification and verification of autonomous robotic systems: A survey. *ACM Computing Surveys (CSUR)*, 52(5):100, 2019.

[5] Carlos Azevedo and Pedro U Lima. A gspn software framework to model and analyze robot tasks. In *2019 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pages 1–6. IEEE, 2019.

[6] Michael Georgeff, Barney Pell, Martha Pollack, Milind Tambe, and Michael Wooldridge. The belief-desire-intention model of agency. In *International workshop on agent theories, architectures, and languages*, pages 1–10. Springer, 1998.

[7] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: Probabilistic symbolic model checker. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 200–204. Springer, 2002.

[8] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A storm is coming: A modern probabilistic model checker. In *International Conference on Computer Aided Verification*, pages 592–600. Springer, 2017.

[9] Jonathan Bohren and Steve Cousins. The smach high-level executive [ros news]. *IEEE Robotics & Automation Magazine*, 17(4):18–20, 2010.

[10] Elvio Gilberto Amparore, Gianfranco Balbo, Marco Beccuti, Susanna Donatelli, and Giuliana Franceschinis. 30 years of greatspn. In *Principles of Performance and Reliability Modeling and Evaluation*, pages 227–254. Springer, 2016.

[11] Vittorio Amos Ziparo and Luca Iocchi. Petri net plans. In *Proceedings of Fourth International Workshop on Modelling of Objects, Components, and Agents (MOCA)*, pages 267–290, 2006.