PN-RTE, Petri net Robot Task Execution

Pedro Miguel Pina dos Santos
pedro.p.santos@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal
May 2016

Abstract

This work presents a software framework which allows the execution of a robot task defined by a formalism based on Marked Petri nets. The proposed solution is composed by three components: a Petri net execution, a primitive action manager and a predicate manager, which address the problems of decision-making, actuation and perception. By implementing and integrating the framework via the Robot Operating System, and withdrawing advantage of its concepts and functionalities, a modular and flexible solution was achieved, capable of realtime execution of different tasks, including hierarchically defined cases. Additionally, the framework provides simple and clear methods definitions, verifications and indepth informations which allows for simple and fast debugs of the task during execution. Finally, the idea that supported the solution architecture implementation and development allows each package to be replaced, extended or used in different scopes.

Keywords: Autonomous Agents, Petri nets, Robotic Systems, ROS, Task Execution, PN-RTE

1. Introduction

Robots are being developed to accomplish complex sets of different assignments in dynamic, partially observed and unpredictable environments. It is clear that to be able to perform complex behaviours properly, a robot must be able to perceive and interact with the environment. Most robots employ high-level decision layers which deliberate or evaluate the sensed environment and through a robot task decide the action or set of actions to execute to achieve the expected goal. However, the design of a robot task plan that could perform a given task in a correct, intelligent and feasible way is a common problem which has been addressed in the literature and can be separated in three main approaches:

1. Manually written, directly programmed in the robot and tailored to the tasks, without using any formal or explicit representations.

2. Manually written based on task representation formalisms.

3. Automatically generated based on the description of the goals and capabilities of the system.

The first approach is highly limited to the expertise and free will of the programmer responsible for the implementation of such tasks, leading to task plans with few actions or too complex or confusing implementations. The second approach is limited by the expressiveness of the representation formalism and capabilities of the designer, although most formalisms offer systematic and consistent modelling methods, which not only allow to verify and/or ensure a task is efficient and feasible, but also leads to richer task models, pruning design errors. The last approach consists in automatic task planning and, although it is more advantageous, the necessity to express all features of interest, allied with the complexity of real applications, introduces a limitation to the application of such approach. Additionally, the tasks obtained from this approach can be expressed by the representation formalisms described in the second approach.

This paper concerns the problem of the execution of a robot task representation formalism framework based on the second approach, describing the implementation of a software framework, based on the Robot Operating System (ROS) [1], for execution of a robot task plans represented by Marked Petri nets. The proposed software framework is available in a public git repository [2] and was tested using both simulated and real robot scenarios.

2. Background

Traditionally, task plans are represented and implemented using discrete events system based approaches, as Finite State Machines [3], or Petri Nets [4]. Finite State Machines are a mathematical model composed by discrete states and transitions between states. In a Finite State Machine, each state represents a unique representation of the
world, evolving from one state to the others through
transitions driven by the execution of actions or ac-
cording to inputs or events received. Application of
such approach can be found in most robotic com-
petitions teams as [5][6] and as a ROS package,
SMACH [7]. Petri nets are a powerful mathemat-
cal and graphical modelling language widely used
for design, model and analysis of discrete event sys-
tems [8]. Petri nets formalism allows to graphi-
cally model aspects such as synchronism, parallel-
ism, concurrency and have a larger modelling
power when compared to Finite State Machines.

2.1. Petri nets
A Petri net consists of weighted, directed, bipartite
graphs composed by a set of places, transitions and
arcs. According to [16], a Marked Petri net is a
eight-tuple (P,T,Pre,Post,M), where:

- \( P \), is a finite, non-empty, set of places;
- \( T \), is a finite set of transitions;
- \( P \cap T = \emptyset \);
- \( Pre : P \times T \), is a matrix which represents the
  set of arcs from places to transitions, such that
  \( Pre(p_i, t_j) = 1 \) if there is an arc from \( p_i \) to \( t_j \), or 0 other-
  wise;
- \( Post : T \times P \), is a matrix which represents the
  set of arcs from transitions to places, such that
  \( Post(t_i, p_j) = 1 \) if there is an arc from \( t_i \) to \( p_j \), or 0 other-
  wise;
- \( M = [m_1, \ldots, m_n] \), is the marking of the net
  and represents the state of the net, where \( m_n = q \)
  means that are \( q \) tokens in a place \( p_n \);
- \( M_0 \), initial marking of the Marked Petri net.

An example of a Marked Petri net is depicted in
figure [1] where graphically:

- **Places** are represented by circles,
- **Transitions** are represented by filled rectan-
gles,
- **Arcs** are represented by arrows,
- **Tokens** are represented by dots inside places.

2.2. Framework Robot task plan representation by
petri nets

Beyond the formal definition of a Marked Petri
net, and from the standpoint a robot task plan
based in Petri nets [9] and in [17], an extension the
Marked Petri net model is described, where place la-
beles are used to distinguish between different types
of places, such as: regular, predicate, action and
task. These places do not introduce changes on the
Marked Petri nets definitions, but increase the anal-
ysis and the design power of a robot task plan.

The following definitions, from [17], describe the
properties of each kind of place:
• A regular place is a normal Petri net place,

• A predicate place represents a logic predicate value, has prefix "predicate." or "p."

• An action place represents a primitive action, has prefix "action." or "a."

• A task place acts as a Marked Petri net macro place, which is used to create hierarchical Petri nets. Has a prefix "task." or "t."

For a better understanding on these definitions, the Marked Petri net in Figure ?? was extended to represent an example of a Robotic Task plan, Figure 2.

Figure 2: A robot task plan modeled by a Marked Petri Net.

2.3. Frameworks to execute and plan a Robot Task on ROS

ROS is currently the de-facto standard choice for the research and development of robotic applications, having a vast collection of packages that implement different robot functionalities, namely robot task plan execution as SMACH and PNP-ROS packages which enable the design of a robot task plan represented by Finite State Machines and Petri nets, respectively.

SMACH is a Python API that allows to design complex robot tasks/behaviours, based on hierarchical state machines, providing two main interfaces: State, which represent state of execution and is defined by its execution and outcomes, and Container, which represent collections of one or more states that implement a specific execution policy. SMACH states can be composed hierarchically, allowing to create complex models and need to be extended by a developer.

A finite state machine representing a possible SMACH implementation of a task equivalent to the one presented in Figure 2 is depicted in Figure 3, where ovals represent states and outcomes are portrayed as arrows with the outcome’s name.

Although the representation in Figure 3 is simpler when compared to the Marked Petri net in Figure 2, it is important to understand that they do not provide the same amount of information. While the framework proposed by [17] explicitly describes the composition of the task using the different places definitions, allowing to quickly understand in which conditions each transition can be enabled in terms of the task model, in SMACH those details are deeply immersed in the implementation of the user.

The PNP-ROS package implements a bridge between the Petri net Plans library and ROS. Petri Net Plans are modelling language that formally defines a robot task plan by a set of elementary structures: no-action, ordinary action and sensing action, and the combination among those structures using control structures (operators), such as sequences, loops, interrupts and concurrent execution operators.

According to [10], a Petri net Plan is a Marked Petri net where: places represent execution phases of actions: initiation, execution, termination, transitions represent events, may be labelled with conditions that control their firing and are grouped according to categories: action starting, action terminating, action interrupts and control, all arcs have a weight value of one.

An equivalent task modeled by a PNP to the one depicted in Figure 2 is shown in Figure 4.

The model obtained is not only different in terms of the places and transitions labels and properties but also in the way the Petri net model is executed. First, transitions are the elements responsible for the control and execution of the actions. Second, transitions are event based, which means that an
enabled transition only fires based on the evaluation of the associated event. Finally, the formalism is only partially implemented on ROS, up to date, relying on an external library in order to execute the Petri net model and the knowledge base.

3. Petri net Robotic Task Execution

PN-RTE, Petri net Robot Task Execution, is a ROS framework capable of execution a Marked Petri net model that represents a robot task plans and the actions present on each task. PN-RTE is divided in three different packages each concerning a different scope of the execution:

- Petri Net Execution, responsible for parsing, storing and executing the Marked Petri nets,
- Primitive Action Manager, responsible for controlling the execution of primitive actions;
- Predicate Manager, responsible for storing, handling and managing the logical predicates.

While Petri Net Execution and Primitive Action Manager were developed and implemented from scratch, for the Predicate Manager the decision was to use a solution already available in the ROS repository, [18, 19].

3.1. Architecture

The packages were developed, and integrated, as C++ programming libraries for the ROS Hydro Medusa version [20], using the ROS environment and the BOOST 1.48 C++ Libraries [21, 22].

There was concern of having separated packages according to each functionality, allowing to minimize the complexity of each package and to maintain internal details of each as a black box to the others.

In terms of communication between the packages, the solution was achieved using ROS Topics. The strategy about the topics and the respective implementation are described in Subsection 3.2.

A scheme of framework showing where each package stands in terms of a robotic component is depicted in Figure 5.

3.2. Framework Communication

Communication between packages was achieved using ROS Topics, that are a message transport layer abstraction over the Transmission Control Protocol [23], and follows the communication protocol defined in the Predicate Manager, which already had two topics defined, /predicate_updates and /predicate_updates, and three messages types that minimize the size of the messages in order to promote efficiency and avoid network congestion.

Additionally, each package was implemented in order to be able to handle possible failures, alerting the user whenever an irrecoverable failure occurs.

3.3. Petri Net Execution

The Petri Net Execution Package is the core package of the PN-RTE framework since it is the one responsible for parsing, validating and executing all the Marked Petri nets models according to the values of predicates received and responsible for broadcasting which actions should start or stop in each moment. The package follows the requirements defined for the framework and is composed by four data structures, Parser, PetriNetStructure, PetriNetExecutor and PetriNetManager.

The division among the four data structures was based on keeping the implementation of package, and the respective functionalities, more clear and modular to further modifications or upgrades. While the first three data structures regard a single Marked Petri net, the last data structure regards the manager which creates, stores and calls the execution of each Marked Petri net when needed. The flowchart of Figure 6 displays the hierarchy between each data structure and the most important incoming and outgoing connections between the packages or data files.

Figure 6: PetriNetExecution Package structure and communication

The Parser is a struct responsible for the parsing of the Petri net data from a pnml file to a raw data structure. It is composed by several methods that
allow a valid transcription of the data to a set of local data structures.

The PetriNetStructure class implements containers in order to store the immutable data that represents a Marked Petri net, namely the set of places, set of transitions, both pre and post matrices and the initial marking. It is important to notice that pre and post matrices were implemented using a combination of two dictionaries of keys since pre and post matrices are usually sparse and this allows to reduce the memory usage.

The PetriNetExecutor class defines the methods required for the execution of a Marked Petri net. Keeping the execution methods separated from the PetriNetStructure data allows to store only the execution representation of the Marked Petri nets that are actually running. This class implements an executeNet method that is responsible to execute a Marked Petri net that represents a robot task, evolving from one marking to another by firing enabled transitions at a time, until a state where there are non active transitions is reached. In case more than one transition is enabled, one, and only one, is randomly chosen.

The PetriNetManager class is defined in order to allow the development of hierarchically Marked Petri nets that represent a robot task plan. The class implements two methods, addNet and executeNet, based on a recursive Depth-first search algorithm that starts in the top layer robot task plan. It is also the class that ensures the communication with the Primitive Action Manager and the Predicate Manager.

3.4. Primitive Action Manager
The Primitive Action Manager is a library for handling the execution of primitive actions, starting and stopping them according to the received information from the Petri Net Execution package. The diagram of the Primitive Action Manager depicted in Figure 7 illustrates what needs to be defined and registered by the user, the connection between the different elements of the package and the connection to ROS or other external packages. The three different arrow colors, black, orange and blue, represent communication/connection to ROS or external packages, registration of the user defined primitive actions by the user and the actions execution call from the manager, respectively.

User defined primitive actions are implemented through the use of C++ classes, each usually representing a single primitive action, existing a clear association between a class instantiation and the start of an action, or destruction and stop.

The manager class also establishes a communication protocol with the Petri Net Execution Package in order to notify which actions were registered and receive the set of actions that should be executed at each step.

3.5. Predicate Manager
The Predicate Manager, part of the Markov Decision Making metapackage developed by Joo Messias during his Phd thesis [24], is a library that allows a user to create and register predicates based on logical conditions, and to publish a set of updates whenever their logical values change. Similar to the Primitive Action Manager diagram depicted before, the diagram of the Predicate Manager depicted in Figure 8 illustrates what needs to be defined and registered by the user, the connection between the different elements of the package and the connection to ROS or other external packages. Once again, the different arrow colors, black and orange, represent communication/connection to ROS or external packages and the registration of the user defined predicates by the user, respectively.

4. Experiments and Results
In order to demonstrate the proposed framework several Proof of Concept tests were developed and executed using a simulated and a real scenario.

In order to ease the development of the tests, and since the focus is in testing the framework integration and do not require a complex simulation environment, the decision was to take advantage of the mdm_example package present in MDM metapackage and modify it in order to implement the developed solution.

Additionally, the models used to perform the tests were designed in order to be easy to understand but powerful to exemplify different execution paradigms and how the framework behaves in each situation. Note that, although all models are based
in patrol tasks, the proposed framework is not restricted to model this kind of problems.

Some of tasks models used to test and evaluate the proposed framework are:

1. Patrol with two conflict situations;
2. Patrol with concurrent actions;
3. Hierarchical task;

4.1. Simulated Scenario

The mdm_example package scenario is composed by a map based on the blueprint of the eight floor of the North Tower from the IST campus, and by a physical description of a nonholonomic robot, Pioneer3-AT [25], with a laser range finder on the front of the robot. The simulation uses Stage [26] as the physical engine, which is a two-dimensional simulator that provides cheap and fast computation models.

Additionally, the package provides the configurations for the localization and guidance, through the amcl\textsuperscript{1} and move_base\textsuperscript{2} ROS packages, and an implementation of several nodes of the MDM_LIBRARY which were removed in order clean up the package from unnecessary nodes.

Thereafter a set of tasks that are described in the following Subsections were designed, a node of the sound play package was added in order to provide audio output capabilities to the robot and not only a Primitive Action Manager node was implemented but also additionally needed predicates were added to the Predicate Manager node. The actions implemented are divided in two sets: Navigation actions, which publish a message with a specific goal pose to the move_base goal topic, and Text-to-Speech actions, which publish specific message with a text string to the sound play.

- **Actions:**
  - go2LRM, − go2Elevator,
  - go2CoffeeRoom, − go2SoccerField,
  - ttsCurrPosition, − ttsLRM,
  - ttsElevator, − ttsCoffee,
  - ttsMoving.

- **Predicates:**
  - IsNearLRMPosition,
  - IsNearElevator,
  - IsNearCoffeeMachine,
  - IsInSoccerField,
  - IsMoving,
  - IsNearStairs.

\textsuperscript{1}http://wiki.ros.org/amcl
\textsuperscript{2}http://wiki.ros.org/move_base

Figure 9 shows the navigation goal positions of each navigation action as filled circles and the areas that trigger a change in the localization predicates as squares. The correspondence between the respective colors and the actions/predicates are: Yellow → LRM, Green → Elevator, Blue → SoccerField, Orange → CoffeeRoom and Purple → Stairs.

Figure 10: Non-deterministic patrol task between four positions.

Figure 11: Map with navigation the goal positions and areas that trigger changes in the predicates.

4.2. Patrol with two conflict situations

The patrol task depicted in Figure 10, which includes conflict situations, is composed by four different actions that are executed individually. The actions which the robot should perform change according to the predicates that evaluate if the robot arrived to the goal position of the respective action.

4.3. Patrol with concurrent actions

Additionally, the framework was tested using a patrol task that has concurrent actions. For this case, a new task containing text-to-speech and navigation actions was created, with action *ttsCurrPosition* having a capacity of one defined, and where both the predicate *IsNearStairs* and its opposite *NOT IsNearStairs* are used in order to avoid a model that would generate infinite tokens. The graphical representation of the achieved patrol task is depicted in Figure 12.

Figure 12: Once more, Figure 13 displays the path followed by the robot while executing the task. Notice that,
while the action go2Elevator was already being executed when ttsCurrPosition starts, the actions go2CoffeeRoom and ttsMoving start at the same time.

Please notice that if the task was changed by modifying the place ttsMoving to go2SoccerField, two concurrent actions that use the same physical actuator were supposed to be executed at the same time, leading to an erroneous task execution, for now this issue can only be identified by the task designer, however a possible solution based on classes of actions is proposed in the future work.

4.4. Hierarchical Task
For the simulator scenario the last task created was a simple hierarchical patrol task, where top level evaluates if an user requests the presence of the robot in the soccer field or if the robot should perform the simple sequential patrol task from Figure 14 using it as a task place, as it is depicted in Figure 15.

For the implementation of this task plan a new predicate userRequest was added to the Predicate-Manager in order to check if an user has requested the presence of the robot in the soccer field area, being true if a request has arrived and false otherwise. Once again, the execution of the task is displayed in Figure 16.

4.5. Real robot scenario
After the validation of the implemented framework using simulated examples, the framework was tested using a MBOT, a four wheel omnidirectional drive robot, on an updated version of the map used in simulation, in order to cope with the more recent layout of the space.

The framework was installed in the robot and the previous described scenarios were used. The robot was able to perform the tests correctly from the framework standpoint, obtaining differences in the randomness choice of actions for some tasks, the
execution time of the navigation actions and the trajectory of the robot, as expected.

Lastly, a new task was designed and tested. The task consisted in a patrol between four positions, but instead of the framework being executed completely in the robot, it was executed from an external computer and the robot was only used to perform the actions and sense the environment. The Marked Petri Net of the task is displayed in Figure 17, while the actions and predicates were implemented in a similar fashion to the ones described for the simulated scenario and are relative to positions of a smaller area of the previous depicted map.

Figure 17: Marked Petri net model of the new Patrol task.

As expected the framework was executed properly and the robot path for this task is represented in Figure 18.

Figure 18: Representation of the robot path while executing the new patrol task.

5. Achievements

This paper proposes a framework for the execution of robot tasks described by a marked Petri net [17] for the Robot Operating System, called Petri Net Robotic Task Execution, PN-RTE. The framework is composed by three packages, PetriNetExecution, PrimitiveActionManager and PredicateManager, where the two first packages were developed and implemented from scratch and the last one was integrated.

There are several nuances of the PN-RTE framework relative to the SMACH and PNP-ROS. PN-RTE allows to execute a robotic task represented by Marked Petri nets which are richer and have an inherent larger modeling power when compared to FSM’s approaches. PN-RTE allows to execute hierarchical tasks and concurrent actions, extracting those directly from the Marked Petri net model.

Additionally, PN-RTE provides a modular framework approach, completely developed inside the ROS environment, where the task executor is clearly detached from the primitive action and the predicate managers, allowing to, not only, separate packages between devices if needed, exchanging only the minimal data required, but also to correct implementation errors in actions or predicates without terminating the other packages.

Finally, the framework was tested in simulated and real scenarios. First by modifying the example scenario from the MDM_Library and using it with several Proof of Concept robot tasks, proving the framework correct execution and demonstrating the principal functionalities and the results from the execution, and lastly using a real robot to perform the previously defined Proof of Concept robot tasks.

6. Future Work

With the current formalism and framework implementation there is no verification on the type of actions that are executing concurrently, which means more than one action could be actuating over the same physical component at the same time leading to execution errors or unexpected results. A possible solution is using classes, or types, of actions using a similar approach to the one present in the framework [17] for differentiat places, using for instance a suffix to represent each action type.

The last subject is related to the problem of the multi-robot representation. While the software framework proposed can be used for single or multi-robot execution, since one could model communication and synchronism procedures using action and predicates places and running the framework of each robot, the formalism presented in the framework [17] does not allow to represent multiple robots in the same Marked Petri net, which would allow to easily model communication and synchronism for the multi-robot tasks. This problem could possible be solved applying other extensions of Petri nets formalism, as for instance colored Petri nets.

References


