Petri Nets for Modeling and Coordination of Robotic Tasks

Pedro Lima, Hugo Grácio, Vasco Veiga, Anders Karlsson*
Instituto de Sistemas e Robótica, Instituto Superior Técnico — Torre Norte
Av. Rovisco Pais, 1; 1096 Lisboa Codex; PORTUGAL
E-mail: pal@isr.ist.utl.pt

Abstract

Petri nets have been widely used to model dynamic systems, namely manufacturing systems. In this paper we introduce the use of Petri nets to model robotic tasks. Different views of the robotic task model can be modeled by distinct Petri net types: interpreted Petri nets for task design and execution, generalized stochastic Petri nets for task quantitative performance evaluation and ordinary Petri nets for task qualitative performance evaluation. Quantitative performance evaluation and improvement based on reinforcement learning from feedback are detailed in the paper. Examples of applications to visual servoing and catching of moving objects by a robotic arm and to mobile robot tasks are presented.

1 Introduction

Petri nets have been widely used to model dynamic systems [1], notably automated manufacturing systems [7]. They share the properties of more general discrete event dynamic systems [6]. Those facts turn Petri nets into good candidates for *qualitative* performance evaluation of robotic tasks. What is singular with Petri nets is their ability to provide simultaneously means for *quantitative* performance evaluation, as well as to allow interaction between an operator and the task under execution.

This paper introduces a new framework under which Petri nets are used for qualitative and quantitative performance evaluation, as well as a tool to design and execute robotic tasks. This framework is an extension of previous work by Wang and Saridis [8], where Petri nets were first proposed as models of robotic tasks. Later, Lima and Saridis [3] introduced a methodology for robotic tasks performance evaluation and learning-based improvement through feedback, which is mapped here to Petri nets.

In Section 2 we introduce our robotic task model, as well as the different Petri net types used to model different views of that model. Section 3 tackles the issue of task quantitative performance evaluation. Learning through feedback the translations of places into primitive actions, which optimize the performance evaluation function, is the subject of Section 4. A tool to design and execute interpreted Petri nets suited for the coordination of robotic tasks, is introduced in Section 5. Two examples of applications to robotic systems are described in Section 6. The paper ends with some preliminary conclusions and projects for future work (Section 7).

2 Petri Net Views of a Robotic Task Model

A robotic *task* is defined in [3] as a string of *primitive tasks*, representing the sequence of actions the robotic system must carry out to accomplish the task goal. Each primitive task may be actually implemented by more than one *primitive action* (e.g., a locate object primitive task can be implemented by a set of different image processing algorithms, defined here as primitive actions). Primitive tasks and their translating primitive actions must be established at design time, associated to specific goals (e.g., to locate an object, to follow a trajectory). When, during the execution of a primitive task, its specific goal or an error state (e.g., due to a timeout) is reached, an *event* occurs and must be detected. Events can be internal or external, depending on whether they are detected within a primitive task or by a device or sensor not checked by a primitive task. To reach its goal, a task must first reach the specific goals of each of its com-

*The work of this author was supported by a grant of the Swedish Government. He is currently a student of the Lund Institute of Technology, Sweden, doing his final year project at ISR/IST.

posing primitive tasks.

Primitive actions, primitive tasks, tasks and events constitute a *robotic task model*. One can look from different viewpoints at such a model. Different Petri net types [1] are used depending on the viewpoint taken. The following subsections illustrate this concept.

Task Design and Execution

The actual task implementation (i.e., its design and execution) requires the scheduling of the primitive tasks composing the task, as well as the synchronization with external and internal events. Events are crucial to coordinate task execution, as they signal when a primitive task can be called for execution, either after another primitive task has finished its job or synchronized with other primitive task(s) execution. An interactive man-machine interface is also important, so that the appropriate schedule of primitive tasks can be designed and task execution can be followed and/or modified by an operator.

Interpreted Petri nets are used to model task implementation. Under our framework and within this type of Petri nets, places represent resources, such as a robot, an object to be manipulated or a primitive task under execution. Whenever a token is inside a place representing a primitive task, this means that the primitive action chosen to translate the primitive task is running. Events are linked to transitions. An internal event linked to a transition occurs when a primitive task associated to one of the transition input places determines that its specific goal has been reached. An external event is detected by devices or sensors whose output is not checked by any primitive task (e.g., a crossfire sensor detects the presence of an object). An external event may always become internal, should it start being detected by a primitive task. Nevertheless, there are cases where keeping an event external is important.

At design time, places and transitions must be linked by the task designer such that the robotic system goes through the desired sequence of specific goals that must be reached before the task goal is accomplished. The designer must also identify all the resources other than primitive tasks required at each task step, and represent them by places. He/she must also provide, for each place associated to a primitive task, two output transitions: one corresponding to a successful completion of the primitive task, another to an exit upon an error situation. In the latter case, an appropriate error recovery procedure must be specified. To avoid a cumbersome task representation, the error recovery branches may be hidden in the graphical display of the Petri net associated to the task.

During task execution, a transition is *enabled* if each of its input places have a sufficient number of tokens available. This happens when all the required resources are available and all the primitive tasks associated to those places are running. However, the transition will only be *fired* when all the events linked to it occur. The tokens are then deposited in the output places of the transition, enabling the execution of their associated primitive tasks. An operator may follow task execution by following the token flow through the Petri net representing the task.

Quantitative Performance Evaluation

Once an Interpreted Petri Net has been designed to represent the actual task implementation, one may evaluate quantitative properties of the task performance by modifying its associated Petri net, turning it into a *generalized stochastic Petri net*.

Generalized stochastic Petri nets can be used to model time-related properties (such as the average task execution time or primitive task bottlenecks) and/or task *reliability*, defined as the probability that the task will meet its specifications, i.e., that it will achieve its goal [3]. Reliability is actually estimated on line during task execution, by measuring the relative frequency of trials where task specifications were met.

Task execution time can actually be determined by associating time to places (P-timed model). The time assigned to each place will determine the performance measure obtained afterwards. For instance, if the CPU time taken by the primitive tasks associated to each place is used, the total CPU time spent by the task will be computed. One may use the actual time taken by each primitive task instead. In this case, the actual time taken by the task will be computed. Of course, this will be a stochastic variable, but random times can be associated to the places under the P-timed Petri net model. When those times are distributed according to an exponential law, the marking of this stochastic Petri net is an homogeneous Markovian process [7], whose well known properties help to determine the time properties of task execution.

Random switches [7] are used to model task reliability. They are associated to the two alternatives at the exit of a primitive task, *normal execution* or *error recovery required*, referred in the previous subsection. Error and normal execution probabilities are associated to those switches for modeling purposes. The probabilities can be estimated by measuring the relative frequency of primitive task executions ending with or without error, respectively.

Qualitative Performance Evaluation

Ordinary Petri net models (or some of their abbreviations [1]) can be used to evaluate qualitative properties [7] of a task, such as *boundedness* (somewhat related to *stability*), *properness* (related to the possibility of error recovery) and *liveness* (indirectly related to *controllability*). Again, if a qualitative performance evaluation is required, the original interpreted Petri net modeling task execution can be modified into an ordinary Petri net (e.g., no event synchronization) to determine such properties.

3 Quantitative Performance Evaluation

A cost function to determine task performance from the performance measure of each of its composing primitive tasks and actions has been introduced in [3]. Such a cost function is general enough to be applied to the diversity of primitive tasks which may compose a robotic task model. It is based on a conjunctive definition of cost and reliability (see [3] for analytic details), which essentially states the following:

- reliability is the probability that a primitive action will meet its specifications (e.g., percentual deviation from a trajectory, overshoot at the end of a point to point motion). This can be measured on line by computing the relative frequency of trials where specifications are met, and corresponds to the probability assigned to the random switch leading to the normal execution branch of the generalized stochastic Petri net view referred above;

- cost refers to CPU time or actual execution time of the primitive action (e.g., the number assigned to the places in the P-timed generalized stochastic Petri net referred above) when the scenario to which the primitive action is applied corresponds to the worst case (i.e., lowest) reliability among those with values lower-bounded by some target reliability.

Given some target reliability for the primitive task, the cost of obtaining that reliability can be determined for each of the translating primitive actions, according to the cost measure defined for the problem. Conversely, establishing at design time different cost values for a primitive action will correspond to different reliabilities for the primitive action.

In general, cost increases with reliability. For instance, to improve the reliability of locating a point within a noisy image with a given accuracy, one has to average several pictures of the image. If the cost is defined as the number of pictures needed, it will depend on the target reliability. However, if the number of pictures is established at design time, the reliability will depend on the number of images (i.e., the cost) used to compute the average.

Therefore, a minimum exists, corresponding to the optimal primitive action[1], for the following cost function:

$$J = 1 - R + \rho C \tag{1}$$

where R is the reliability, C the cost and ρ a weight factor such that $\rho C \in [0, 1]$. In general ρ will be such that the cost does not overwhelm the reliability when directing the search for the optimal action. A typical ρ is given by $\rho = \frac{1}{\max_{a \in A} C(a)}$, where A is the set of primitive actions. Note that the cost is computed *a priori*, but in general it can assume any value and may have any units, depending on the primitive task. ρ is used to normalize both the cost value to the interval $[0, 1]$ and the cost units across primitive tasks.

Note that the definition of cost and reliability refers to primitive actions. However, their values, and consequently those of the cost function, can be propagated to the primitive tasks and the task using appropriate expressions [3]. Therefore, the quantitative performance evaluation is extended to the complete robotic task model.

4 Learning the Optimal Translations

We have already pointed out the existence of alternative translations for a primitive task, i.e., each time a primitive task is ready to be executed, the first step consists of selecting which of its translating primitive actions will effectively run. Different alternatives will have different performances, measured by

[1]This is a discrete space optimization problem, hence the minimum found will correspond to the optimal solution among the set of available primitive actions for the primitive task.

the cost function introduced in the previous section. Therefore, it is important to create a mechanism to: i) update at each step the primitive action cost function estimates; ii) learn over time the optimal selection, i.e., the primitive action which minimizes the cost function.

Our framework distinguishes between three primitive action status, returned by the primitive action upon completion: success, when the specifications were fully met, failure, when the specifications were not fully met, but task execution may proceed along the *normal execution* branch, and error, when the specifications were not met and error recovery is required (e.g., the primitive task exited on timeout).

The success and failure signals are used to update the reliability (therefore the cost function) estimates iteratively, after the execution of each primitive action. An error status is interpreted as a failure, for learning purposes, but in this case task execution does not follow its regular path. The availability of success and failure signals suggests the use of a reinforcement learning scheme [2, 4, 5] which updates a probability distribution function over the discrete set of alternative primitive actions for a primitive task. The scheme is proved to converge with probability one to the selection of the optimal action with probability 1 [3]. In practice, this means that, after a number of learning steps, dependent on the number of primitive actions and their cost function values, the optimal primitive action will be chosen with high probability. Therefore, the scheme represents a practical way of using learning to provide adaptation to the environment state and improve performance of a robotic task.

Learning may be slow in some cases, but those correspond to a set of alternative primitive actions with similar performances. Hence, the overall performance will not be considerably affected by the duration of the learning process.

5 A Tool for Task Design and Execution

A Petri-net-based application to coordinate the execution of robotic tasks and provide a man-machine interface has been developed. A robotic task can be designed through a graphical interface, by drawing the corresponding Petri net and associating primitive tasks to places and (when appropriate) events to transitions. Task execution can be followed in real time through the same graphical interface, by following the token flow in the Petri net. The operator can change the task execution path and/or timing by token removal/insertion in special places, used for task flow control only (e.g., step-by-step execution is possible).

The application runs on a hardware architecture based on a virtual multi-processor machine, composed of several PCs linked by a local fast Ethernet network. Each of the processors has also multitasking capabilities.

The software architecture is based on a client-server philosophy. Each PC in the network behaves either as a server or as a client, depending on the circumstances. When acting like a server, a PC provides services, which are applications resident in that server. Services may be divided in primitive actions and general-purpose applications. The latter include functions to communicate between PCs using sockets (TCP/IP protocol), functions which access the global memory of the system, libraries of math functions, board drivers and others. Some of the services are only available locally, i.e., can only be requested by local processes, while others exist specifically to serve requests from other network nodes — which will then behave as clients.

From the designer standpoint, the distribution of primitive action services by processors in the network is transparent, i.e., he/she must initially define in a file the location of the different primitive actions and then the software will know where to direct a request for such a service, each time it is invoked. Data/primitive action requests between network processors are handled by socket-based communication services, always running in every PC of the network. Nevertheless, a wise procedure consists of distributing primitive actions according to the hardware resources allocated to each processor. As an example, in a visual servoing and object catching application of the architecture, described below, motion control primitive tasks (therefore, all their primitive actions) should be located in the PC which directly interfaces the PUMA controller, while image processing primitive tasks and actions should be located in the PC interfacing the camera.

6 Examples of Applications

At the time of writing this paper, we were in the process of using the application described in the previous section to recast the implementation of robotic tasks our group has been working with in the last few years.

Two of those tasks are described in the following subsections.

Visual Servoing and Catching

Based on the distributed control architecture described in the previous section, a testbed for research on visual servoing and catching of moving objects was developed, including a CCD camera located above a table and 2 PCs linked by the local Ethernet network. One of the PCs interfaces the CCD camera through an ISA bus card. PUMA motion control algorithms run on the other PC, interfacing the PUMA through ISA bus cards. The image processing PC sends set points periodically to the motion control PC using TCP/IP sockets. The set points represent the predicted catching points.

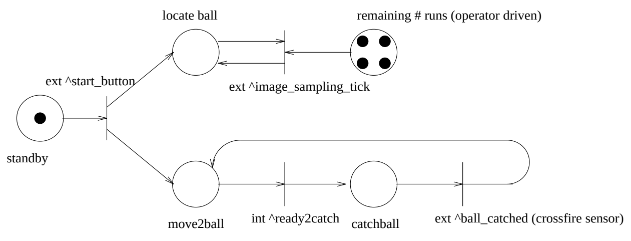

Figure 1: Petri net representation of the visual servoing and object catching task, with initial marking corresponding to a 4 runs experiment. The 'int' and 'ext' label prefixes represent internal and external events, respectively.

An example of an interpreted Petri net representing a visual servoing and object catching task by a robotic manipulator, currently running in our Laboratory, is depicted in Figure 1. The manipulator must catch, at its closest end of the table, rolling ping-pong balls. An individual throws the ball at the other end of the table towards the manipulator. The vision subsystem tracks the ball motion and extracts its relevant motion parameters. From this information, the prediction subsystem iteratively estimates the ball trajectory and the coordinates, in the manipulator base frame, of the point where it will leave the table. The manipulator motion controller moves the manipulator gripper, with the correct orientation, to this point and, equipped with a cup, catches the ball.

An operator may interact with task execution by setting the number of times the task will run autonomously. This is accomplished by token removal from or insertion into the remaining number of runs place in the example.

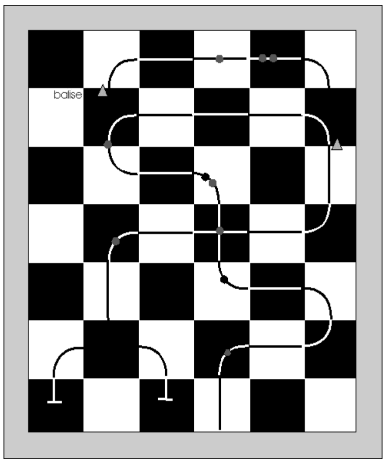

Figure 2: The mobile robot competition track.

In the figure, some of the events are internal, others are external. The latter might become internal, but in some cases this might cause problems. For instance, the ball catched event might become internal to the catchball primitive task. However, should the image sampling tick event become internal to the locate ball primitive task, this would prevent the control of the number of runs by the operator.

Mobile Robot Task

We have been participating in the last few years in a robotic competition, held in France. The objective is to build a robot whose task consists of following a 5cm wide track painted on a chessboard-like surface, composed of 2m side squares of alternating black and white colors. The track has the opposite color of the corresponding background square (see Figure 2) and is composed of 2 meter straight lines and one-fourth of a circle arc segments with 1 meter radius. There are track interruptions somewhere along the path, obtained by replacing the corresponding background square by one with the same color but with no track segment painted on. The robot must detect the interruption, and recover the track at the closest segment, for instance by following a cylindrical retroreflector, located 1 meter above the point where the track resumes. The end of the track is signaled by a T-shaped pattern.

A more complex interpreted Petri net is required to model the robot task in this example. It is depicted in Figure 3. In this case all events are internal. Notice that there are several conflicts in the Petri net, but they all correspond to events linked to boolean variables whose value is changed within the precedent primitive tasks. The conflicts correspond always to a binary choice dependent on the variable value.

As in the previous example, neither error recovery branches nor the primitive actions associated to each primitive tasks are represented in the figures.

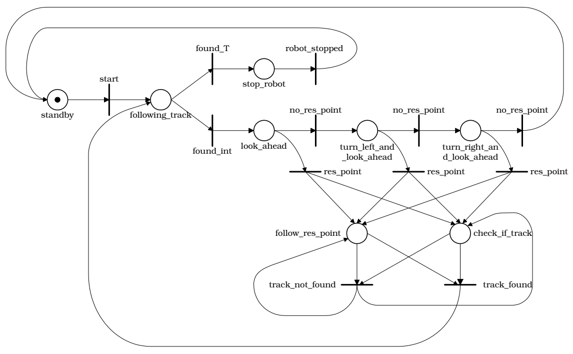

Figure 3: Petri net representation of a mobile robot task.

7 Conclusions and Future Work

This paper introduced some novel concepts regarding robotic task modeling by Petri nets, namely the use of different Petri net types to model distinct views of the robotic task model, as well as the task performance quantification and the use of learning to improve performance over task execution.

One of the issues not yet tackled is the qualitative performance evaluation of robotic tasks using ordinary Petri nets. Boundedness, properness and liveness properties, which can be determined from the ordinary Petri net model, can be mapped to robotic task properties, such as stability, error recoverability and controllability. We intend to clarify those relations in the near future.

Quantitative performance evaluation concerning time-related properties is another topic to be developed. Even though the cost measure referred in the paper is based on the task execution time, so far only the reliability related issues have been studied in more detail. We also believe that more specific time-based cost measures can be used and intend to further study this topic.

Finally, even though the mapping between the robotic task model and the Petri net has been defined, an analytical formulation of this mapping would be desirable, as well as a clear definition of primitive tasks and events. The latter would improve significantly the design of complex robotic tasks.

References

[1] R. David and H. Alla. Petri Nets for modeling of dynamic systems. *Automatica*, 30(2):175–202, 1994.

[2] K. S. Fu and J. M. Mendel. *Adaptive, Learning and Pattern Recognition Systems: Theory and Applications.* Academic Press, 1970.

[3] P. U. Lima and G. N. Saridis. *Design of Intelligent Control Systems Based on Hierarchical Stochastic Automata.* World Scientific Publ., 1996.

[4] K. Najim and A. S. Poznyak. *Learning Automata: Theory and Applications.* Pergamon Press, Oxford, 1994.

[5] K. S. Narendra and M. A. L. Thathachar. *Learning Automata — an Introduction.* Prentice Hall, 1989.

[6] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1), 1989.

[7] N. Vishwanadham and Y. Narahari. *Performance Modelling of Automated Manufacturing Systems.* Prentice Hally, 1992.

[8] Fei-Yue Wang and G. N. Saridis. Task translation and integration specification in Intelligent Machines. *IEEE Transactions on Robotics and Automation*, RA–9(3):257–271, 1993.