A Concurrent Simulator for Petri Nets Based on the Paradigm of Actors of Hewitt

Luca Bernardinello and Francesco Adalberto Bianchi

Dipartimento di Informatica sistemistica e comunicazione
Università degli studi di Milano-Bicocca
Viale Sarca 336 U14, Milano (Italy)
luca.bernardinello@unimib.it

Abstract. In this paper we propose a concurrent simulator for Petri nets based on the model of Actors of Hewitt. The classes of Petri nets that are supported for the simulation are Place-Transition Nets and Elementary Nets. The simulator is written in Scala, a programming language with a library implementing the Actors model.

Keywords: Petri Nets, Simulation, Actor Model, Scala programming language

1 Introduction

This paper deals with the design and implementation of a concurrent simulator for Petri nets based on Hewitt's Actor model. Different versions of the simulator have been built: two versions for Marked Graphs, one version for Free Choice Nets and finally the simulator for general Petri Nets. All the versions support both the firing rule of Elementary Nets and the firing rule of Place-transitions Nets. In this paper we will only describe the case of Place-Transition nets.

The simulator is based on the algorithm described by Dirk Taubner in [4], and is written in Scala, an object-oriented language built over Java. We chose to use the Actor model for the implementation of the concurrent simulator to make a comparison between the level of concurrency offered by this paradigm and that offered by Java threads. The algorithm is explained in Section 2.

The paradigm of Actors was introduced in 1973 by Carl Hewitt in [1]. An Actor is an independent entity which, concurrently to other Actors, can receive messages from other Actors, change its internal state, send new messages and create new Actors. Each Actor is identified by a mailing address; an Actor needs to know the mailing address of the Actors with which it wants to communicate. Each message contains the address of the sender.

Section 3 describes how Scala implements the Actor model. Several experiments have been made whose results are discussed in Section 5.

2 The Simulation Algorithm

Starting from Taubner's analysis ([4]) we have built a simulator such that two concurrent firings in the model correspond to (potentially) concurrent activi-

ties in the program. The strategy applied in our simulator provides a process for each place and for each transition of the net. Each transition communicates with its input places to check the firing condition. If enabled, the transition asks its neighbouring places either to decrement or to increment their number of tokens. The processes associated to places manage their number of tokens and the requests made by transitions. The communication between place and transition must follow a specific protocol because of conflicts. In a conflict situation a place must manage requests from different transitions; for this reason, a token reservation strategy is needed. Conflicts can also generate deadlocks in a simulator; if two transitions have made some of their reservations but they need a further reservation that they can't make because of the reservations of the other, a deadlock situation is generated. To avoid it, a transition must be able to cancel previous reservations.

The strategies just described are applied in the *Polling of places in a fixed order algorithm* (also called PTO). Each transition sequentially polls the processes corresponding to its input places in a fixed order and waits for the answer. At the first refusal, the transition cancels reservations made earlier and restarts polling from the first place. If the transition receives affirmative answers from all the places, it informs each place to increment or decrement its number of tokens. On the other hand, a place manages the reservation requests: if it can satisfy a request, it reserves the necessary token to the transition and it sends an affirmative message to the transition; if it can not satisfy a request, it sends a negative message to the transition.

3 Scala and Actor Model

Several programming languages realize concurrency through the paradigm of Actors. Scala has been designed in 2001 by Martin Odersky and it was released for the first time in 2004 on Java platform. Scala runs on the Java Virtual Machine, providing a high compatibility with existing Java code. Scala is a pure object-oriented language, supporting also functional programming. A library is provided implementing the Actor model. Each Actor in Scala has a mailbox in which it stores the received messages. The `react()` and `receive()` methods pick a message from the mailbox and check if it matches one of the patterns specified in the Actor. Scala Actors using `react()` are lightweight in comparison to Java threads. In particular, if an Actor implements the `receive()` method, Scala makes a one to one mapping with Java Virtual Machine threads; therefore each Scala Actor is implemented as a Java thread. For this reason, a program implemented through Scala Actors using `receive()` method has similar performances of a program based on Java threads. An Actor using the `react()` method instead, when it is started its behavior is captured in a closure and its stack is discarded. At this point, the Actor is suspended and the associated thread is free to execute other tasks. When the Actor receives a message, the corresponding closure is executed by a thread. Using this strategy, a thread is able to execute more than one Actor at the same time.

4 Implementation of the Simulator

The simulator is composed first of all by the `Net`, `Place` and `Transition` classes that define the structure of the net. When the user asks for the simulation of the specified Petri net, the class `Net` provides for the creation of the Actors needed. Two kinds of Actors are needed: one to represent places and one to represent transitions. For this reason, the classes `PlaceActor` and `TransitionActor` have been implemented. Each implements an Actor whose behavior reflects the task described in Taubner's algorithm.

In the first version for Marked Graphs, in which conflicts are not allowed, the reservations and cancellation strategies are not needed. In a second version of the simulator of Marked Graphs each place is seen as just a communication channel between two transitions, and not as an Actor.

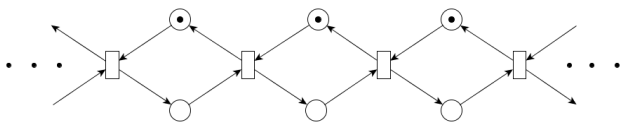
Another version simulates Free Choice Nets, in which if two or more transitions are in conflict, then they have a single input place (that is the same for all the transitions in conflict). In this case the reservation strategy is needed, but there is no need for cancelling reservations.

Finally, the simulator for general Petri nets has been implemented. This version reflects Taubner's algorithm, implementing both reservation and cancellation strategies.

5 Experimental Results

A series of experiments has been made in order to compare the degree of concurrency offered by the simulator based on Actors and the simulator based on Java threads: by degree of concurrency we mean the number of processes that the simulator can start during a simulation. We recall that when we speak about simulator based on Java threads we mean a simulator implemented through Scala Actors using `receive()` method. Moreover we have studied the ratio between the number of transition firings and the number of cancellations of reservations.

For the first experiments we used a net with a regular structure, whose size is simply parameterizable. In particular, we considered the following net, where the basic module can be replicated:



The experiments have been conducted on two different computers: the first has a dual-core processor with 2 GB of memory, the second has a quad-core processor with 4 GB of memory. The results of experiments made on first computer are the following:

	Processes	Core 1	Core 2	Memory used
Actors	18 000	30%	30%	2000/2000 Mb
Threads	200	100%	100%	1200/2000 Mb

From the table it can be seen that the number of processes started in the case of Actors is much greater. In the case of Actors the number of started processes is limited by memory, which is totally occupied; in case of threads, the limit comes from the processors usage. The experiments made on the second configuration have yielded results consistent with those just described.

The aim of the second type of experiments was to record some significant parameters to test the efficiency of the implementation. The net used in this case models the dining philosophers (see Fig 1). Specifically, we consider a variant of the model in which some philosophers take first their left fork, while the others take first the right fork.

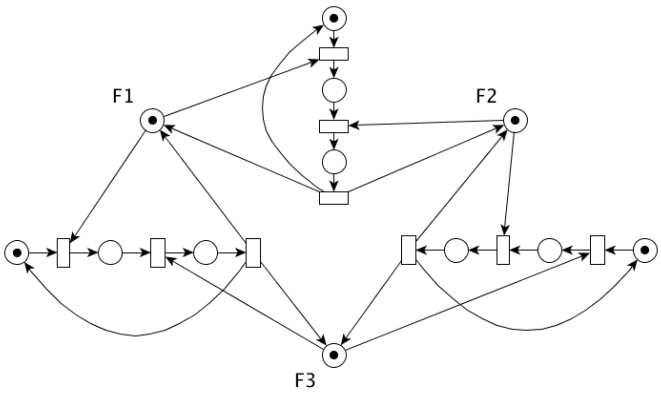


Fig. 1. The net of three dining philosophers.

The first kind of data we wanted to record was the number of transitions that fire and the number of negative answers that the transitions receive.

An interesting property shown by data concerns the ratio between the number of transitions that fire and the number of refusal received by transitions (see table below); this ratio is constant and independent from time and number of transitions. Its value is close to 0.1, which means that each transition, to obtain the permission to fire, receives on average ten refusals. This result is justified by the fact that each transition continually tries to fire, this suggests to explore changes to the algorithm in order to reduce the number of refusals.

Time (sec)	Philosophers	Transitions	Negative answers	Ratio
60	50	2344	22564	0.1038
	250	2357	22678	0.1039
	1000	2355	22421	0.1051
600	500	23620	227666	0.1037

The second kind of data that we have recorded is the number of transitions that fire and the number of messages sent by transitions to cancel a reservation (see the table below). We recall that the number of negative answers (recorded in the first experiments) and the number of cancellations of reservations are not the same. In fact if a transition receives a negative answer from the first place, it does not need to cancel any reservation. In this case the ratio between the number of transitions and the number of cancellations of reservations is approximately 1, which means that each transition sends on average a single message to cancel reservations before firing.

Time (sec)	Philosophers	Transitions	Delete reservation	Ratio
60	50	2332	2359	0.9885
60	250	2344	2252	1.0408
60	1000	2320	1868	1.2419
600	500	23635	24139	0.9791

6 Conclusion

Taubner's algorithm allowed us to implement a concurrent simulator for Petri nets. Furthermore Scala has proved to be a very simple and elegant programming language, which, thanks to Actors model, offers a level of concurrency significantly higher than threads Java. Planned future developments are the implementation of a GUI for the simulator and other tools for analyzing the results of simulations. We will also explore alternative algorithms which exploit structural properties of the net to be simulated. In particular, we will consider nets that can be decomposed in several State Machine. An Actor will be associated to each sequential component.

References

1. Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.
2. Martin Odersky, Lex Spoon, Bill Venners. *Programming in Scala.* Artima, First edition, 2008.
3. Philipp Haller, Frank Sommers. *Actors in Scala.* Artima, 2012.
4. Dirk Taubner. On the implementation of Petri nets. In *European Workshop on Applications and Theory of Petri Nets*, pages 418–434, 1987.