Specific Proposals for the Use of Petri Nets in a Concurrent Programming Course

João Paulo Barros
Instituto Politécnico de Beja,
Escola Superior de Tecnologia e Gestão
Rua Afonso III, n.º 1
7800-050 Beja, PORTUGAL
+351 284 311 543
jpb@estig.ipbeja.pt

ABSTRACT

Concurrency is a difficult subject to teach and learn. This paper presents a set of recipes for the use of Petri nets as a teaching aid of some fundamental concurrency concepts, in the context of an introductory concurrent programming course. Classroom experience clearly demonstrates this usage of Petri nets improves students understanding of concurrency concepts.

Categories and Subject Descriptors

K.3.2 [Computers and Education]: Computer and Information Science Education --- *Computer science education*; D.2.2 [Software Engineering]: Design Tools and Techniques --- *Petri nets,* D.2.4 [Software Engineering]: Software/Program Verification --- *Formal method, Model checking, Validation*; D.1.3 [Programming Techniques]: Concurrent Programming;

General Terms

Documentation, Languages, Design, Verification.

Keywords

Introductory course, FSP, LTSA, process algebra, state machines.

1. INTRODUCTION

Concurrency courses have a very broad variability. They can be taught in different places in the curriculum, typically in operating systems or programming language courses, but also in software engineering courses and distributed databases courses. They can be taught to novice students but, more frequently, to intermediate or advanced students. They can use different tools and languages. Yet they all have in common the necessity of teaching the fundamental concurrency concepts. In our experience, most students, and particularly the ones already trained in sequential programming, have significant difficulty in visualizing concurrent systems behaviour. To overcome this, we soon convinced ourselves that a multiple view approach to system modelling would be the one that better could assist student understanding. Besides we believe that a concurrent programming course, due to concurrent programming intrinsic difficulty, is probably the best place to convey the necessity of good software engineering practices, namely the ones directly related to the software life cycle. These include: analysis, design, verification and/or testing in the design phase, good coding practices and project documentation. Fortunately a book that applies these ideas already exists [3], and we have based our concurrent programming course on it since 1999. For each section, the book follows the sequence: concepts, models and implementation. It clearly and efficiently presents modelling and verification techniques using a simple process algebra language. The language is named FSP (Finite State Processes) and was developed for pedagogical purposes. It also contains an associated tool for specifying and analysing the FSP models expressed in textual form. Starting from the FSP model, the tool (named LTSA for Labelled Transition System Animator) is capable of graphically display the processes as state machines. It also generates the state machine resulting from the processes parallel composition. It allows progress and safety properties verification, as well as model execution through the use of an interactive simulator where, in each execution step, the user selects one of the enabled actions. This usage is proposed in the context of a software life cycle encompassing analysis, design, verification and coding. Finally, it also proposes a simple but effective technique to translate FSP models to concurrent programs in the Java Programming language.

The approach followed in the book has proven quite effective. Yet we have found that student learning, namely the understanding of some fundamental concepts can be significantly improved by the simultaneous presentation and use of Petri net models [5].

2. THE COURSE

The course follows the referred book [3] which is used as the main reference. As such, in each course section we also present concepts prior to models and these prior to coding. We present UML class and object diagrams, FSP and Java concurrent constructs. Java is used due to its current popularity and included support for concurrency, namely monitors.

2.1 Assessment

Student assessment is based on a semester long project and an open book written exam.

The project implies analyse, design and coding of a simple concurrent system containing a few processes. These are coded as Java Threads. Students are required to use UML, FSP, Petri nets and Java. We also encourage the generation of project documentation, namely UML class diagrams and source code documentation.

The written exam presents some simple concurrent systems and asks for FSP specifications and/or Petri nets models. It also explores safety and liveness properties.

2.2 Course Structure

Each course section has the following structured sequence:

1. Concepts
2. Relevant FSP constructs.
3. Relevant Petri net constructs.
4. Construction of the FSP model and analysis of its behaviour using the LTSA tool.
5. Construction of the Petri net model from the FSP specification and/or from the state machines generated by the LTSA tool.
6. Simulation of the Petri net model.
7. Translation to a UML class or object diagram.
8. Coding of the Java program.

Points 3, 5 and 6 constitute the proposed additions to the methodology in [3].

3. THE USE OF PETRI NETS

As presented in the previous section, Petri nets are presented along the course, together with the corresponding FSP constructs. In this way, students are able to understand the similarities of both notations and also the advantages and disadvantages as applied to each concept. Most notably students are forced to separate the concepts from their expression in one particular modelling tool. So that the overhead of learning yet another notation is no obstacle to the intended course objectives, low-level Petri nets were preferred against other Petri nets classes, namely high-level nets [2], which, although more powerful, would require too much course time. Ordinary Petri nets also have the added and very significant advantage of mapping very nicely to FSP models. In addition a simple and friendly Petri net tool was used: Visual Object Net++. This and many others can be found in the Petri nets tools database [4]. Students react very favourably to the use of Petri net tools that allow the graphical editing and simulation (token-player) of Petri net models. Based on student reactions, we are absolutely convinced that the use of at least one tool is necessary for a good and motivating study of Petri nets. Next sub-sections present, in detail and by means of example problems, the proposed usage of Petri nets. This usage emphasizes the relation between Petri nets and FSP models.

3.1 Synchronization and Deadlock

Synchronization is clearly visualized in almost any Petri net. Here we propose the use of Petri nets to visualize a deadlock initial state. This section uses the FSP model from an exercise proposed in [3] as an example of the added advantage on the use of Petri nets for better understanding of concurrency concepts. In this case the concepts are synchronisation and a deadlock resulting from a "wait-for cycle". The FSP model, containing three primitive processes ("Alice", "Bob" and "Chris") and one composite process ("S"), is the following:

```
Alice = (call.bob   -> wait.chris -> Alice).
Bob   = (call.chris -> wait.alice -> Bob).
Chris = (call.alice -> wait.bob   -> Chris).

||S = (Alice || Bob || Chris) /{call/wait}.
```

Each necessary synchronisation is specified through the use of the same action (same name) in more than one state machine. The operator / specifies sets of pairs for action name substitution. In the example above, "wait" is substituted by "call". That implies, for example, that "wait.chris" will be replaced by "call.chris", specifying, in this way, synchronization between processes "Alice" and "Bob". The LTSA tool translates each process modelled in FSP to a graphical representation of a state machine (Figure 1).

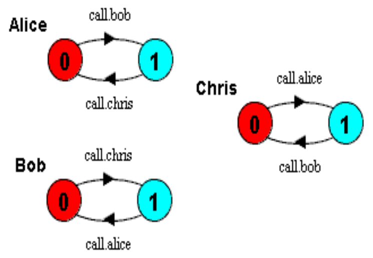

Figure 1. State machines generated by the LTSA tool.

The tool can generate composite processes resulting from the parallel composition of those state machines. Yet, in this particular example, the deadlock does not allow that visualization, as the initial state is a deadlock state. This fact is not immediately obvious. In fact the explanation of this situation is proposed as the exercise to the student.

The Petri net model allows a much more intuitive perception of the dependencies (synchronisations) among processes. For that we ask the student to translate the state machines into Petri nets and group the synchronising actions, which have become Petri net transitions (Figure 2). This is a very simple process quickly grasped by students. Them we merge the grouped transitions and obtain the composite process model as a single Petri net (Figure 3).

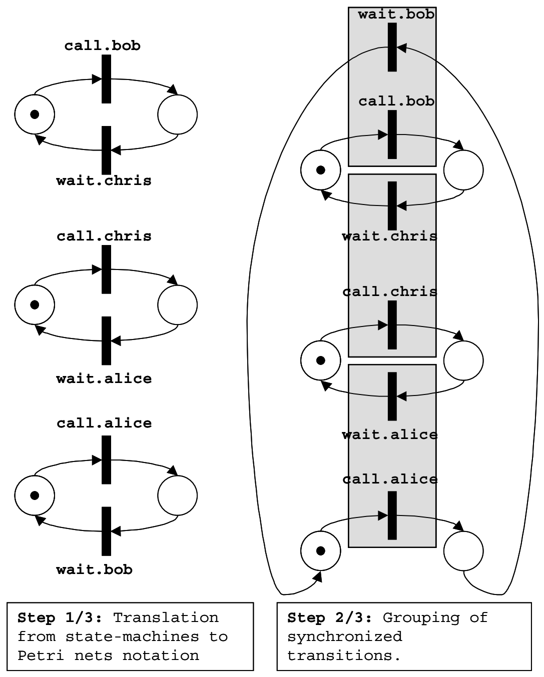

Figure 2. From parallel state machines to Petri nets. The figure illustrates the first two of a total of three steps.

A brief look at this Petri net makes clear that the system is deadlocked as no transition can fire. This results from each transition dependency on an unmarked place on the right.

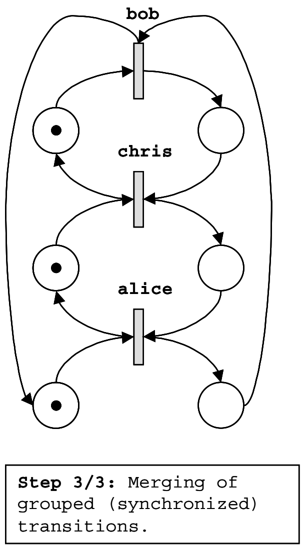

Figure 3. From parallel state machines to Petri nets. The figure illustrates the last of a total of three steps.

Next we present a more elaborate example emphasizing process composition understanding.

3.2 Process Composition

Process composition can lead to some perplexities due to unobvious action synchronisation. Petri nets uncover these complexities in a visual and dynamic way. The following example illustrates just that.

Consider the following FSP model that specifies a simple semaphore with states red and green.

```
CLOCK = (tick -> CLOCK).

CLOCKL(N = 2) = (start -> LG[0]),
LG[lg:0..6] = (when (lg < N) tick -> LG[lg+1]
               |when (lg == N) long_tick ->
               CLOCKL).
|| LONG_CLOCK = (CLOCK || CLOCKL).

SEMAPHORE_GREEN = (change_to_red ->
                    SEMAPHORE_RED),
SEMAPHORE_RED = (change_to_green ->
                  SEMAPHORE_GREEN).
```

The semaphore is controlled by a process “CLOCK”, that generates an action named “tick”, and a “CLOCKL” process, that generates a “long_tick” action after two “tick”. The process “LONG_CLOCK” is a composite of “CLOCK” and “CLOCKL”. The “SEMAPHORE_GREEN” process specifies the light change. The timing is to be imposed by the composition of “SEMAPHORE_GREEN” with one or more “CLOCK” and/or “LONG_CLOCK” processes. The semaphore should change to red after one “tick” in green and change to green after one “long_tick” in red.

A first attempt could be:

```
||SEMAPHORE_TIMED = (SEMAPHORE || LONG_CLOCK)
                    /{change_to_red/tick,
                     change_to_green/long_tick}.
```

Unfortunately, this does not work. The “change_to_red” action will consume one “tick” that should be used by the “change_to_green” (“long_tick”). As such the semaphore will be one single “tick” in the red state instead of two. To avoid this problem we can try to add one CLOCK process to the composite:

```
||SEMAPHORE_TIMED = (SEMAPHORE_GREEN ||
                     LONG_CLOCK ||
                     CLOCK)
                    /{change_to_red/tick,
                     change_to_green/long_tick}.
```

This has a new, more subtle, problem: we have one more process “CLOCK”, besides the one “inside” “LONG_CLOCK”, but it remains synchronized with the “LONG_CLOCK”. As such we have not made any progress yet. This is much more obvious in the Petri net model (Figure 4).

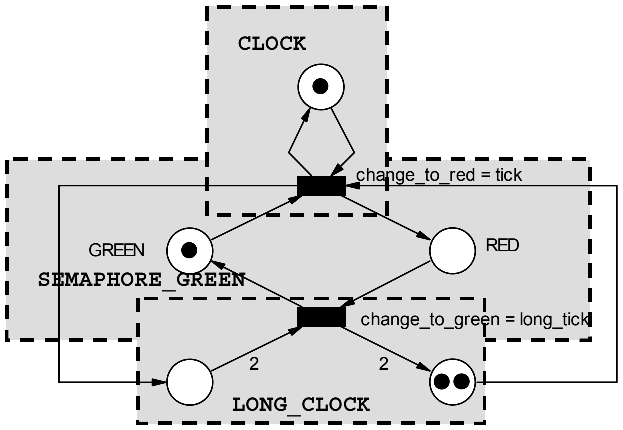

Figure 4 Petri net for the FSP model where CLOCK and LONG_CLOCK synchronize in the action tick.

It is enough to observe that after the firing of the transition associated to actions “change_to_red” and “tick”, the net will deadlock as no transition will be enabled. Students can observe this behaviour using the mentioned Petri net tool, or any other tool that allows token-game graphical simulation, will be adequate.

At this point most students will realize by themselves that the problem is the “use” of process CLOCK by process LONG_CLOCK. From this a possible solution is quite obvious: LONG_CLOCK should use a different CLOCK (labelled “a” in the following code):

```
||SEMAPHORE_TIMED = (SEMAPHORE_GREEN ||
                     LONG_CLOCK ||
                     a:CLOCK)
                    /{change_to_red/a.tick,
                      change_to_green/long_tick}.
```

The corresponding Petri net is even more readable (Figure 5). Yet, its simulation, reveals that the independency between “CLOCK”s allows the evolution of “LONG_CLOCK” before the semaphore gets to “RED” state. Namely, a “tick” can occur while not yet in the “RED” state. That allows the transition to “GREEN” after only one tick in the “RED” state. To avoid this we must start “LONG_CLOCK” on transition to RED:

```
||SEMAPHORE_TIMED = (SEMAPHORE_GREEN   ||
                     LONG_CLOCK ||
                     a:CLOCK)
                  /{change_to_red/{a.tick,
start},
                    change_to_green/long_tick}.
```

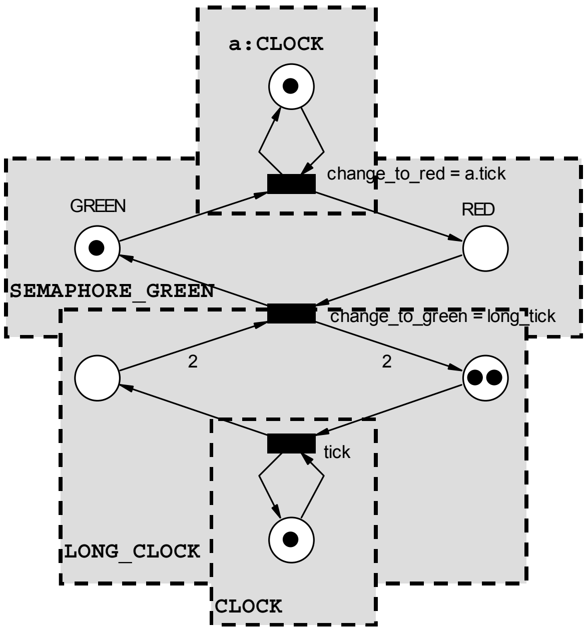

Figure 5 Petri net for FSP model with one more clock.

The final Petri net only starts the LONG_CLOCK process on arrive to state RED (Figure 6).

Note that the composite state machine, generated by the LTSA tool, contains all the interleavings allowed by the synchronisations. This is very useful to understand the concept of interleaving and, by consequence, the state explosion problem, but it has the cost of making unclear the dependencies between processes, namely their synchronisations. Even the parallelism is not clear as the interleaving hides it under the states representing combinations of more than one state in the original state machines. By contrast, a Petri net allows a clear visualization not only of the initial state machines (which are just a particular case of a Petri net) but also of the parallel composition and respective synchronisations.

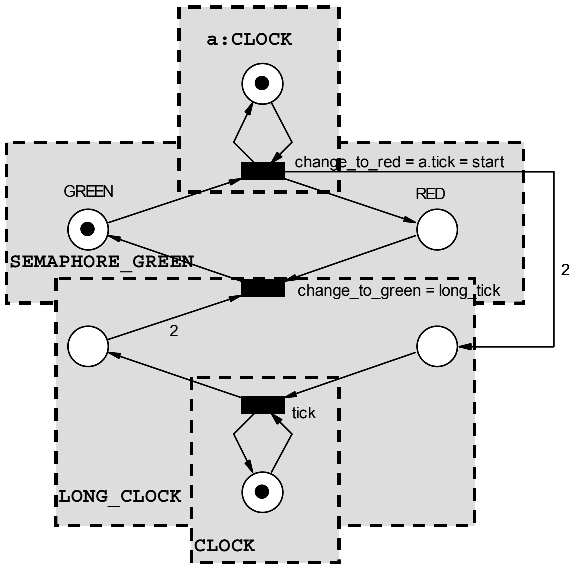

Figure 6. Petri net for FSP model with clock restart on change to RED.

3.3 Conflict and Starvation

Petri nets provide an extremely readable model of the classic dining philosophers problem [1]. Figure 7 presents a simplified specification that avoids deadlock by imposing an atomic acquisition of the respective forks by each philosopher. The concepts of conflict and starvation are absolutely clear in a Petri net simulation. We can almost “see” the philosophers trying to get their forks. A simple modification allowing the acquisition of each fork in separate, by each philosopher, also allows deadlock visualization

4. EVALUATION

The presented examples were used in one edition of the referred introductory concurrent programming course; two previous editions had used preliminary versions of those same examples. In the course we had 16 students. About half the students had one year experience using the Java language in an academic setting; the others had zero or negligible experience programming Java or any other object-oriented programming language. Most of them had been exposed to concurrent programming only in an operating systems course through the study of trivial concurrent programs written in the C programming language and using UNIX system calls.

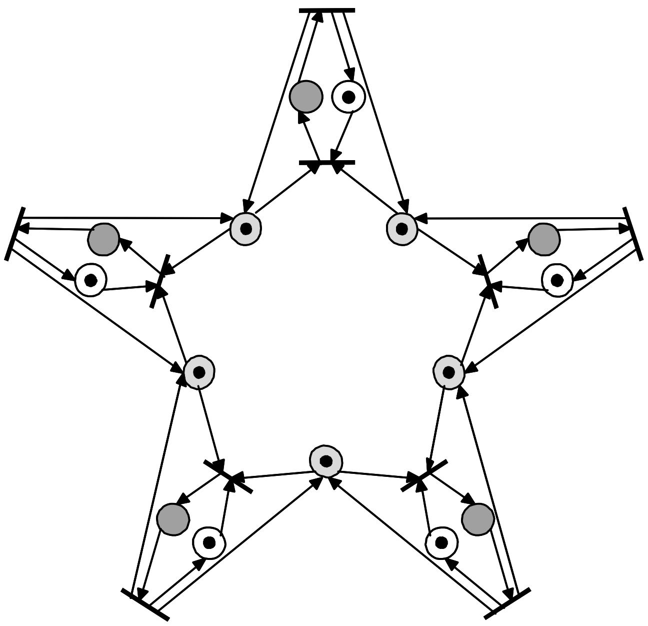

Figure 7. Petri net for the dining philosophers problem.

So as to get more rigorous feedback about the use of Petri nets a questionnaire was given to the 16 students; 10 of them answered. The questionnaire used a Likert scale: from 1 (strongly disagree) to 5 (strongly agree). The answers to the questions relevant to the evaluation of the presented Petri net usage are summarized in Table 1.

Table 1. Answers to the questionnaire given to students.

Question	**Average in a Likert scale (1-5)**
Petri nets allowed a better understanding of process composition	4.3
Petri nets allowed a better understanding of the starvation and deadlock concepts	4.4
Overall, Petri nets allowed a better understanding of concurrent systems operation	4.4
The construction of FSP models PLUS Petri nets, helped the understanding of concepts and concurrent systems	4.4
Petri net models are easier to understand than FSP models	3.8
Petri net models are easier to build than FSP models	3.3
It is easier to start by doing the Petri net model rather than by the FSP model	3.7
Petri net models make Java programs easier to understand.	3.8

The inquiry clearly showed that students found the addition of Petri nets useful particularly in the understanding of concepts. They did not found Petri nets models to be easier to build than FSP models, but the majority found easier to start by the Petri net model as opposed to the FSP model. Also, most of the students found that the Petri net model helped them to understand the Java programs. These results confirmed our feedback from the class, namely that the presented usage of Petri nets improves students understanding of concurrent programs and concurrency concepts.

5. CONCLUSION AND FUTURE WORK

We have found the Petri net formalism to be particularly valuable in the understanding of process composition, synchronization, deadlock, conflict and starvation. The presented examples demonstrate their usefulness as an aid in student understanding of these concepts. Students particularly valued the extra insight provided by the Petri nets models and animated simulations. We notice that the understanding of all studied systems always improves when Petri net models are studied together with FSP algebraic models. In this sense we intend to increase the coupling between FSP models and Petri net models. One way to accomplish this will be to create additional examples of Petri net modelling and respective relation to FSP models. Another will be the developing of an automatic translation tool from FSP to Petri nets. This will allow a faster presentation of Petri nets in classroom and significantly help the student when self-studying

6. ACKNOWLEDGEMENTS

The author thanks Luis Gomes, from Faculdade de Ciências e Tecnologia of Universidade Nova de Lisboa, and the three anonymous reviewers for their valuable comments on this paper.

7. REFERENCES

[1] Ben-Ari, M. “Principles of Concurrent Programming”, Prentice-Hall, 1982.

[2] Jensen, K. e Rozemberg(Eds.) *High-level Petri Nets:Theory and Application*, Springer-Verlag, 1991.

[3] Magee, Jeff, Kramer, Jeff, “Concurrency State Models and Java Programs”, John Wiley & Sons, 1999.

[4] Petri Net Tools, http://www.daimi.au.dk/PetriNets/tools/.

[5] Reisig, Wolfgang, “Elements of Distributed Algorithms – Modeling and Analysis with Petri Nets”, Springer, 1998.