Jochen Spranger

FUNlite — A parallel Petri Net Simulator

1 Introduction

In the development of provably error-free control software for manufacturing systems the application of Petri nets is a well-known approach. One of the main advantages of Petri nets are their sound mathematical background which make it possible to analyse and validate the qualitative and quantitative behavior of a Petri net system by formal methods. The gap between the validated Petri net model and the needed control software can be closed by directly synthesizing the control software form a Petri net specification. One way to achieve this, is to assign control code to the transitions and use a Petri net simulator to simulate the tokenflow of a Petri net.

We use save hierarchical Place/Transition nets to develop modularized control software for manufacturing systems [3]. Each machine controller is modelled in a separate subnet. Those subnets are connected by places which reflect the communication between the controllers.

For this kind of Petri nets we have realized a parallel Petri net simulator (FUNlite), especially designed for fast execution speed, low memory consumption and low communication overhead. In FUNlite we generate for each subnet a fast sequential Petri net simulator. Each subnet simulator is mapped on a different processor.

When simulating a Petri net system, the speed of the transition enabling test plays an important role. In FUNlite the enabling test is highly simplified by a method characterizing the enabling of a transition at a given marking by a simple number comparison.

Communication places where a conflict between subnets exists are administrated by the lockset-method [9] which is speeded up by an analogous number comparison technique.

The FUNlite system has been implemented on a Transputer system consisting of T9000 Transputers which are connected by C104 communication switches. The code is written in INMOS C [6] which is an extension of ANSI-C by the CSP model [5] for parallel programming. The main parallel features are processes, synchronous communication through channels and semaphores.

This paper is organized as follows. Section 2 gives a short introduction in Petri nets. Section 3 presents the simulation engine for the subnets. Section 4 and 5 show the internal and external administration of communication places. Finally some conclusions are given.

2 Basic Notations and Definitions

Definition 1 (Petri Nets) *A Petri Net $N = \langle P, T, F, m_0 \rangle$ consists of*

1. *Finite, nonempty sets P and T such that $P \cap T = \emptyset$. Elements of P and T are called places and transitions, respectively.*
2. *A mapping $F : (P \times T) \cup (T \times P) \to \mathbf{N}$.*
3. *A mapping $m_0 : P \to \mathbf{N}$, called the initial marking.*

This kind of Petri net is also called *Place/Transition Net*. As usual we use the following notations:

1. The pre- and postsets of a transition resp. of a place are given by $\bullet x = \{y \in P \cup T : F(y,x) > 0\}$ and $x\bullet = \{y \in P \cup T : F(x,y) > 0\}$,
2. For each transition $t \in T$ the mappings $t^-, t^+ : P \to \mathbf{N}$ are defined by $t^-(p) := F(p,t)$ and $t^+(p) := F(t,p)$.
3. $\Delta t := t^+ - t^-$

A marking of a Petri net is a function $m : P \to \mathbf{N}$, where $m(p)$ denotes the number of tokens in a place p. A transition $t \in T$ is enabled (may fire) at a marking m iff $t^- \leq m$ (i.e. $t^-(p) \leq m(p)$ for each place $p \in P$). When an enabled transition t at a marking m fires, a new marking m' given by $m'(p) := m'(p) + \Delta t(p)$ is reached. A Petri net is called *save* iff $m(p) \leq 1$ for every $p \in P$ and every marking m reachable from the initial marking m_0. Two transitions $t, t' \in T$ are in *conflict* iff they have common preplaces. We use hierarchies as syntactical constructs to structure the Petri net into subnets. Each subnet has a border of places. Those places are called *communication places* because they are responsible for the connection between the subnets. The pre- and postplaces of a subnet are those communication places which are pre- or resp. postplaces of a subnet transition. Two subnets are in conflict iff they have common preplaces.

For a more detailed introduction into Petri net theory we refer to e.g. [8].

3 The Petri net simulator FUNlite

For each subnet (i.e. machine controller) we generate a sequential Petri net simulator. He is divided in a data structure which represents the Petri net, functions of control code which are assigned to the transitions and a simulation engine which plays the tokengame on the data structure. When a transitions gets fired their assigned code is executed. The benefits of a sequential Petri net simulator are:

- the overhead of process switching can get avoided. (With one CPU there is anyway only pseudo-parallelity possible.)
- we do not have to synchronize the execution of transition code to keep it atomar.
- the resolution of conflicts can be done fast and simple.

One of the main problems when simulating a Petri net system is the speed of the transition enabling test. In FUNlite we use a highly simplified enabling test, where the enabling of a transition at a given marking is indicated by a counter.

For each transition a counter is introduced which characterizes the firability of this transition. The counter of a transition t shows the number of unmarked preplaces of t. If the counter of a transition t decreases to 0 the transition gets enabled. After the firing of a transition t, we only have to consider the transitions t, $(\bullet t)\bullet$ and $(t\bullet)\bullet$ to determine the set of new enabled transitions. For each transition t' in $(\bullet t)\bullet$ we increase the counter of t' by the number of common preplaces with transition t. For each transition t' in $(t\bullet)\bullet$ we decrease the counter of t' by the number of common places between the preplaces of t' and the postplaces of t.

```
Integer TransCounter[#Transitions];
Set of Transition EnabledTrans;
Transition t;

SimEngine() {
  Init(TransCounters, EnabledTrans);
  repeat
    t := SelectEnabledTrans(EnabledTrans);
    foreach t' ∈ (•t)• do
      Inc(TransCounters, t');
      if Disabled(t', TransCounters) then
        EnabledTrans := EnabledTrans \ {t'};
      fi
    od
    Fire(t);
    foreach t' ∈ (t•)• do
      Dec(TransCounters, t');
      if Enabled(t', TransCounters) then
        EnabledTrans := EnabledTrans ∪ {t'};
      fi
    od
  forever
}
```

Figure 1: Sequential Simulation Engine

4 Internal administration of communication places

In each subnet we introduce special code to connect the subnets through communication places. We distinguish three different cases.

Post communication places: For each post communication place of a subnet we generate a process and an output channel. The process waits for an activation from the sequential simulation engine and then starts to transmit a token through the output channel. The output channel is connected to the administration process of the corresponding pre communication place. In introducing a separate process for token distribution we get an asynchronous coupling between the sequential simulation engine and the subnet-to-external-place communication. This results in a speed up of the tokenflow animation because the sequential simulation engine could not be blocked by synchronous channel communication. We want to remark that the saveness assumption implies that one process per post communication place is enough to handle the token distribution

Pre many-to-1 communication places: This kind of places can have many presubnets but only one postsubnet. They are mapped to the processor of the corresponding postsubnet. For each place we generate a process which waits on its input channels for the arriving of a token. Each input channel corresponds to an input arc. When a token arrives it gets queued. Before the sequential simulation engine selects an enabled transition for execution

it clears the token queue and updates the transition counters of the subnet. Here we have to remark that saveness assumption implies that there can be no blocking of a process that wants to deliver a token to a subnet

Pre many-to-many communication places: This kind of places can have many presubnets and many postsubnets. There exists a conflict between the postsubnets, such there exists no unique mapping of the communication place to a postsubnet. Therefore they get extern administrated by special processes. For the interaction with those external processes the sequential simulation engine have to be expanded. Transitions with pre many-to-many communication places get (pseudo) enabled if their counters are equal to the number of pre many-to-many communication places. E.g. the firability of such a transition only depends on external places. The code for the selection of an enabled transition in the sequential simulation engine has to be changed as follows:

```
L1: t := SelectEnabledTrans(EnabledTrans);
    if IsManyToManyTrans(t) then
      if AskForTokens(t) == not ok then
        goto L1;
      fi
    fi
```

Figure 2: Selection of an enabled transition

This is the only place in FUNlite where we use a polling technique.

5 External administration of many-to-many comm. places

We have subdivided the many-to-many communication places in disjointed sets L (so-called locksets) [9] such that holds:

- $\forall s, t \in T$ with $\bullet s \cap \bullet t$ contains many-to-many communication places
  $\Longrightarrow \exists l \in L$ with l contains all many-to-many communication places of $\bullet s \cup \bullet t$
- |L| maximal

With the introduction of locksets we obtain a simple conflict resolution and atomar allocation of more than one token. A lockset l is implemented by:

- For each Transition with a many-to-many communication place in l we introduce a counter which represents the number of missing communication place tokens (analogous to the counter technique in the sequential simulation engine).
- For each place p in l we generate an input process which waits for an arriving token on his input channels. If a token arrives it updates the counters of the corresponding transitions.
- An administration process which reacts on token requests from the corresponding subnets.

The changing of the counters by the input processes and the administration process is synchronized by a semaphore.

```
Integer TransCounter[#Transitions];
Transition t;

repeat
  t := WaitForRequest();
  if TransCounters[t] == 0 then
    Send(ok);
    Update(TransCounters, t);
  else
    Send(not ok);
  fi
forever
```

Figure 3: Administration Process

6 Conclusion

We have developed a fast parallel Petri net simulator for save hierarchical Place/Transition nets. It has been designed for fast execution speed, low memory consumption and low memory overhead.

The key techniques are a fast and highly simplified enabling test for transitions and a speeded up conflict resolution on communication places based on the lockset-method. This approach can be easily extended to structurally bounded Petri nets [1].

To evaluate the concepts behind the Petri net simulator we have implemented the FUN-lite system on a T9000 Transputer system. We use our Petri net EDitor PED [2] for the construction of Petri nets annotated with control code. The generation of the code for the Transputer system is fully automated.

As a medium-sized example we have used a Petri net model of a really existing production cell in a metalprocessing plant [7]. The production cell consists of six components that are organized in a (closed) pipeline. The generated control software runs a simulation environment of the production cell implemented with Tcl/Tk. The communication between the control software and the simulation environment is based on a simple Input/Output protocol. We use a master process to interact with the simulation environment to serialize the simple I/O statements from the separate control processes.

In future we plan to design a special intermediate representation of the hierarchical structured Petri net. In that way we will decouple the graphical frontend and the code generation.

References

[1] J. L. Briz and J. M. Colom. Implementation of Weighted Place/Transition Nets based on Linear Enabling Functions. In *Application and Theory of Petir Nets, LNCS 815*, pages 99–118, 1994.

[2] G. Czichy. Design and Implementation of a Graphical Editor for Hierarchical Petri Net Models (in German). Diploma Thesis, TU Dresden & GMD/FIRST, Berlin, 1993.

[3] Monika Heiner and Peter Deussen. Petri Net Based Qualitative Analysis – a Case Study. Techn. Report I-08/1995, BTU Cottbus, Dep. of CS, 1995.

[4] Monika Heiner, Peter Deussen, and Jochen Spranger. A Case Study in Developing Control Software of Manufactoring Systems with Hierarchical Petri Nets. In *1st Int. Workshop of Manufactoring and Petri Nets*, Osaka, June 1996.

[5] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[6] INMOS Ltd., SGS Thomson Microelectronics. *T9000 ANSI C Toolset*, 1994.

[7] C. Lewerentz and T. Lindner. Formal Development of Reactive Systems – Case Study Production Cell. In *LNCS 891*, 1995.

[8] P. H. Starke. *Analyse von Petri-Netz-Modellen.* G. B. Teubner, Stuttgart, 1990.

[9] Dirk Taubner. Zur verteilten Implementierung von Petrinetzen. *Informationstechnik*, 30(5):357–370, 1988.

Autorenangabe:
Dipl.–Math. Spranger, J.
Brandenburgische Technische Universität Cottbus
Postfach 101344
D–03013 Cottbus
Tel: (+49-355)69-3825
Fax: (+49-355)69-3830
E-mail: `jsp@informatik.tu-cottbus.de`