

Using Agents for Simulating and Implementing Petri Nets

Tom Holvoet* and Pierre Verbaeten
Department of Computer Science, K.U.Leuven,
Celestijnenlaan 200A
B-3001 Leuven Belgium
e-mail: Tom.Holvoet@cs.kuleuven.ac.be

Abstract

This paper presents a software architecture for simulating and implementing Petri nets. It is based on object-oriented techniques and autonomous agents. Object-orientation enables the adaptation and extension of the software architecture with new or alternatively defined features. Agents allow to model a net as a set of autonomous, cooperating entities. The result is a flexible and extendible framework of reusable components for efficiently implementing a large family of Petri net classes.

The execution can be performed on a mono-processor, a parallel or distributed system. This is the result of using the XENOOPS execution environments for parallel applications.

1 Introduction

Petri nets are a commonly used formalism for modelling and analysing complex concurrent systems. They make a simple, yet powerful visual formalism for describing concurrency, synchronization, causality and non-determinism between system activities. Besides the fact that Petri nets lean themselves to deriving useful properties (e.g. through place and transition invariants), a Petri net system specification is a good starting point for two other important development activities: simulation and implementation. *Simulation* of a Petri net helps in obtaining basic intuition about the behaviour of the modelled system and enables experimental analysis and (empirical) validation of the model. *Implementing* a Petri net means providing an executable program that behaves exactly as modelled by the specification. Most existing systems or environments that support Petri net simulation and/or implementation are “inflexible” in that design decisions are taken statically, such as the class of Petri nets, the transition rule, transition scheduling policy (how does the system select transitions for checking their enabledness and possibly fire them), and, in the case of a distributed implementation / simulation, the policy that is used for balancing the work load over a set of processors. For some systems, design decisions even influence the kind of Petri net that is provided to application developers.

In this paper we describe the principles for developing a *reusable, flexible and extendible* software architecture for the *efficient, parallel and distributed* simulation and implementation of a large family of Petri net classes, as well as a partial (prototype) realisation. The approach is

based on *object-orientation*, which allows to obtain the software quality characteristics mentioned above, and *agents*, which provide the appropriate abstractions for developing concurrent and distributed applications.

Note that it is not our intention to present a fancy tool with a large amount of functionality for analyzing Petri nets. Instead, our purpose is to apply promising software development techniques for implementing and simulating Petri nets, which could be seen as *the* example of a concurrent application.

2 General Philosophy

2.1 Goals

Support for a large family of Petri net classes

The intention of our approach is to support any class of Petri nets which retains the “local control property” of nets: the firability of a transition depends on its vicinity only (and not on the firability of other transitions, as is often the case for Timed and Stochastic Petri nets). Though we do not pretend to provide an implementation for all classes of Petri nets, we do claim that an implementation of any member of this family of Petri net classes requires a highly reduced amount of manpower.

Efficient implementation and simulation

We want to provide a software architecture which allows an efficient parallel and distributed implementation of a large family of Petri nets. As such, it should be possible to use properties of particular classes of nets in order to reduce execution time.

Software Quality

Flexibility, reusability and extendibility are characteristics of high-quality software. While these have been underestimated in the area of Petri net implementations, we will show the benefits of software which complies to these properties.

Parallel and distributed execution

Simulating a large Petri net is known to be a computationally expensive task. In order to use the available system (a mono-processor, multi-processor (shared memory) or multi-computer (distributed memory)) to its full extend, a suitable implementation should allow parallelism and distribution in a flexible way. A suitable dynamic load balancing policy exhibits a key role for the success of distributed implementations.

2.2 Objects and Agents

The approach we adopt for developing the software architecture for Petri net implementation is based on two

*Research Assistant of the Belgian Fund for Scientific Research

pillars: agents and object-orientation. Developing a system using agents means modeling and realizing the system as a set of autonomous *active objects*. These agents are proactive (they can decide themselves what actions they perform), they act independently and concurrently, and they cooperate through message passing. Passive objects, on the other hand, do not have this autonomous active behaviour, they are merely reactive (they can only respond to messages sent to them, they cannot initiate any action autonomously). Using agents for modeling systems has been accepted as a step forward towards better modeling techniques.

In an agent model, distribution and parallelism is implicit. The concrete reification of the agents and object in a parallel and distribution can be described orthogonally to the logical aspects of the agent model. A suitable environment should hence provide concepts as load balancing, location independent object invocation, and so on.

3 Excerpt of the Software Infrastructure

The backbone of the design of the software architecture is the implementation of an abstract class of Petri nets, consisting of a set of possibly annotated places, transitions, edges and tokens. The dynamic behaviour of these nets is described by a *generic enabling rule*: “if a transition is enabled, it can fire”.

Each particular class of nets can then be described as a specialization of a generic net. Hence, an implementation of a particular class of Petri nets will be modelled (and implemented) as specializations of the components of the generic net implementation (in particular, through inheritance).

The first step in our architecture is the set of four basic classes:

Places A first class of passive objects is **Place**. Instances of this class represent the places in a Petri net. Place objects mainly serve as token containers, which are queried and manipulated on behalf of other objects (e.g. during transition firing).

Edges The **Edge** class represents edges in a Petri net. Edges are conceived as a relationship between a place object and a transition object. Edge objects do not constitute any autonomous behaviour. Their interface represents their reactive behaviour, offering particular functionality to transition objects.

Tokens Tokens are plain passive objects.

Transitions The only class of *agents* in the Petri net implementation is the class of transitions. Transition agents are only proactive: they repeatedly try to fire the transition they represent.

3.1 An Overview

Now we can refine this model as to provide support for an implementation of a number of well-known classes of Petri nets. We therefore build an inheritance hierarchy for each of the components.

Places

We first extend the model by a set of subclasses of the Place class. The inheritance hierarchy (see Figure 1) for places is the result of overviewing several classes of Petri nets, and investigating the functionality of the respective places. Each subsequent subclass imposes an

additional restrictions on places, and each subclass will correspond to places of particular classes of Petri nets. As mentioned, objects of the class Place represent places which can contain any number of any kind of token. The interface operations that are available for TokenPlace objects are: *object constructor*, *AddToken*, *GetTokenOfColor*, *RemoveTokenOfColor*, *RemoveToken*, and so on. Two specialisations of the Place class are the **CapacityTokenPlace** class and the **MonoTypeTokenPlace** class. The former models places that are labelled with a capacity that indicates the number of tokens the place can contain. The type of the tokens remains unspecified. The latter specializes places in another way. As its name indicates, MonoTypeTokenPlace objects contain any number of tokens of one specific type. The type of these places is represented as an attribute to these objects.

Next, we can model places that can only contain a restricted number of tokens of one specific type, represented by the class **SimplePlace**. By a multiple inheritance relation, this class inherits from both the CapacityTokenPlace class and the MonoTypeTokenPlace class, since it models places that possess both properties.

Finally, the class **Condition** inherits from the SimplePlace class, imposing the additional property that the place can only contain one type-less token.

Besides of this hierarchy, which deals with the properties of places concerning the tokens they can contain, another hierarchy, focusing on more implementation-oriented properties, can be modelled. These mainly concern strategies for the order in which tokens can be retrieved from places. Several strategies can be used, such as FIFO (first in first out), random policy, priority based token selection (tokens are assigned priorities), and so on. This is modelled by providing an abstract class **OrderPlace**, which models places that are imposed some strategy for token delivery. Descendants from this OrderPlace class are the classes **FifoPlace**, **PrioPlace**, **RandomPlace**, and so on. If a place in a net in a particular Petri net implementation ought to follow a particular strategy, it should be an instance of a class inheriting from both a token containment class and an appropriate class that realizes the order of tokens.

Consider a place in a Pr/T-net to which a FIFO ordering is imposed (meaning that if multiple tokens can be unified with the arc variables in order to fulfill the corresponding transition predicate, the token(s) that have been in that place for the longest time are selected before the others). Such a place is an instance of the class **FifoOrderTokenPlace**, multiply inheriting from the classes Place and FifoOrderPlace.

Transitions

The behaviour of transitions of any kind of Petri net can be rephrased in terms of these transition objects. These are accordingly modelled as subclasses of the Transition class. It allows to provide a more efficient implementations for the firing protocol and the transition rule if the definition of the net allows so.

Edges

Edges in our model are objects that interconnect a place with a transition or vice versa. Their task is twofold: upon demand of a transition object, they can check

whether they agree on the transition enabledness (e.g. in the case of a C/E-net, a transition input edge should be able to check whether a token is present in the associated input place), and they can shift tokens (either extract tokens and present them to transitions in the case of an input edge, or accept tokens from a transition and shift them through the output places). The interfaces of edge objects are, *mutatis mutandis*: *Convenient* (to check its agreement on enabling the corresponding transition), *RemoveTokens*, and *so on*. The inheritance hierarchy is rather similar to the first part of the place hierarchy and the respective class names should be self-explanatory.

Tokens

Tokens are objects. The type of tokens is represented by an attribute for Token objects. As a special case, anonymous tokens (as in C/E-nets, P/T-nets and so on) are Token objects with attribute value set to “anonymous”. Other, more complicated types of tokens, such as coloured tokens, are instances of subclasses of the Tokens class, which are to be defined by the designer of the CPN.

3.2 Extendibility: Illustrations

Several researchers have proposed extensions to “standard” Petri net definitions for reasons of expressive power or expressive comfort: inhibitor arcs, clearing arcs, transitions that shift tokens to output places depending on preconditions, information arcs, and so on. In this section, we want to illustrate the strength in extendibility and reusability of the presented architecture by showing how “new” Petri net features are easily included.

Figure 1 depicts the class hierarchy of the architecture enhanced with the extensions mentioned in this section.

Inhibitor Arcs

It has been shown that some systems cannot be modelled by Petri nets without introducing inhibitor arcs (e.g. producer-consumer systems with priority [Mur89]). An inhibitor arc is an arc that connects a place to a transition, which has the effect on the net behaviour that the transition is disabled when the place contains a token. To incorporate inhibitor arcs in the architecture, new classes are defined. The abstract base class **Inhibitor** models the basic notion of inhibitor arcs. The **TypeWeightedInhibitor** class that inherits from this Inhibitor class and from the **TypeWeightedArrow** class models inhibitor arcs similar to the ones mentioned above. When a transition queries such a **TypeWeightedInhibitor** object, it checks whether the appropriate tokens are *not* available. The **WeightedInhibitor** class represents inhibitor arcs which prevent a transition to be enabled if the corresponding place contains at least the number of tokens denoted by the arc label. The most traditional kind of inhibitor arc is represented by **TraditionalInhibitor** class, which agrees to the transition enabledness if no tokens are available in the corresponding place. Including an inhibitor arc into a Petri net is straightforward. A transition object does not need an adaption of its implementation. It still maintains a list of input arrows, which can now either be ordinary or inhibitor arcs (polymorphism), and the transition rule remains the same (since it queries the arrows for their

approval according to the enabledness).

Clearing Arcs

Clearing arcs are a special kind of input arcs that withdraw all tokens from the respective input places when the corresponding transition fires. This could result in the classes **TypeClearingArc**, representing objects that withdraw all tokens of all types mentioned in their label from the input place, and **TraditionalClearingArc**, modelling arcs that withdraw all tokens from the input place. Similar to inhibitor arcs, clearing arcs can be included into an implementation or simulation of a net without further changes.

Diversifying Transitions

In [Bas92], diversifying transitions are proposed for modelling actions that do not always end in the same way. Depending on the result of the transition firing, tokens are shifted only to a subset of the output places. We provide a new class **DiversifyingTransition**, which inherits from the transition class and adapts its firing protocol, such that token shifting towards the output places is conditioned by some predicate that can be provided for each output arrow.

4 Distributed Execution: XENOOPS

XENOOPS[JBV93] is an environment for executing agent-based systems on a parallel or distributed computer system. It provides support for

- *active objects* (agents);
- *location-independent object invocation* the physical distribution of objects among several processors is transparent to these objects.
- *dynamic load balancing* a distributed load balancer tries to enhance processor occupation by migrate objects between processors according to a (customizable) policy.
- *fault-tolerance management*
- and so on.

A Petri net implementation or simulation using our architecture consists of a set of communicating objects, some of which are autonomously active. Since XENOOPS is an environment that manages the execution of object-oriented applications, we have XENOOPS manage the Petri net implementation or simulation. This allows to execute a net in a distributed environment.

To obtain an execution that meets the efficiency expectations, we customize the environment, among others, by providing an appropriate XENOOPS load balancer. The problem of distributed allocation of net components can be dealt with in several ways. E.g. one can use a clustering technique which intends to allocate sets of potentially conflicting transitions (“conflict cluster” [Wik90]) and the places influencing their enabling to the same processor. Another strategy uses semi-flows for clustering. Other (not net-related) techniques can also prove their usefulness. The environment presented in [PKV94] allows to detect clusters of closely coupled objects, i.e. objects that have a high degree of mutual invocations on each other. In a net implementation, this would reveal clusters of transitions and places that most depend on each other.

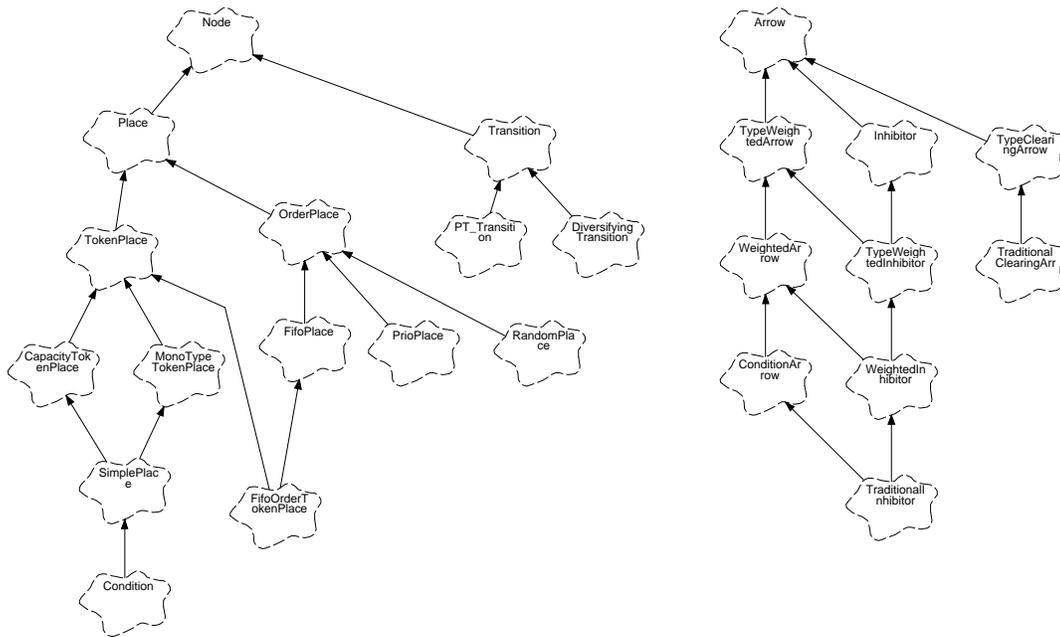


Figure 1: Extended Class Diagram of the Petri Net Framework

5 Conclusion

We presented a software architecture for implementing and simulating Petri nets. The underlying philosophy is quite different from existing systems and approaches: we aim to supporting *any* kind of net, on a mono-processor as well as on parallel and distributed systems. The architecture has been designed and implemented using object-orientation and agents, which appear to be a perfect methodology for developing extendible and flexible complex concurrent systems.

Real-world entities are in fact autonomously active and cooperating substances. Agents are the software abstractions for these entities. Since in the context of nets, transitions are really concurrent and cooperating entities, agents are fit for modelling the transition activity. Moreover, allowing the agents to be mobile (i.e. not bound to one physical processor in a distributed environment) automatically entails distributed simulation and implementation of nets. The dynamic allocation can be described independently (e.g. by a load balancer).

Object-orientation provides techniques, mechanisms such as inheritance, polymorphism and dynamic binding, which prove essential for software extendibility, flexibility and reusability. For our architecture, it allows to provide support for any kind of net and it eases the addition of new net features or the alteration of existing components. Open implementation is used such that algorithmical design decisions (such as firing protocol, enabling rule, and so on) can nicely be changed.

Future work will focus on the tools for developing nets. Tools are responsible for checking consistency of nets (not intermixing condition places with transitions with predicates, and so on). They should also support extensions or adaptations to the architecture, as well as perform run-time simulation, and analysis.

We recently started working on a non-trivial PN-

TOX[HV95] application, which should provide some hands-on expertise for ameliorating our tools and environment.

References

- [Bas92] R. Bastide. *Objets Cooperatifs: un Formalisme pour la Modelisation des Systems Concurrents*. PhD thesis, Universite Paul Sabatier de Toulouse, February 1992.
- [HV95] T. Holvoet and P. Verbaeten. PN-TOX: a Paradigm and Development Environment for Object Concurrency Specifications. In *Proceedings of the Workshop on Object-Oriented Programming and Models of Concurrency, ICATPN'95, Turin, Italy, 1995*. To appear.
- [JBV93] Wouter Joosen, Stijn Bijnens, and Pierre Verbaeten. Object Parallelism in XENOOPS. In *Proceedings of the Object-Oriented Numerics Conference*, April 1993.
- [Mur89] T. Murata. Petri Nets: Properties, Analysis and Applications. In *Proceedings of the IEEE*, volume 77, pages 541–580, 1989.
- [PKV94] W. De Pauw, D. Kimelman, and J. Vlisides. Modeling Object-Oriented Program Execution. In *Proceedings of the European Conference on Object-Oriented Programming '94*, pages 163–182, 1994.
- [Wik90] D. Wikarski. Object Nets - a Canonical Class of Models for Behaviour Simulation and Structure Synthesis of Distributed Systems? In *Proceedings of the Intl Seminar on Modelling, Evaluation and Optimization of Dependable Computer Systems*, pages 91–100, 1990.