Proceedings of the European Simulation symposium 1994
Ed. A.R. Kaylan, A. Lehmann, and T.İ. Ören

OBJECT-ORIENTED TOKENS, A WAY OF INCREASING THE MODELING POWER OF SIMULATION NETS

Åke Gustavson and Aimo Törn
Åbo Akademi
Computer Science Department
SF-20520 ÅBO, Finland
E-mail: agustavs@ra.abo.fi

ABSTRACT

Simulation Nets (SNs) are Petri Nets extended for convenient modelling of discrete event simulation problems (Törn 1991). Experience from modelling and simulation by two interpreting tools suggest that more modelling power is needed. The nets have to be extended in several ways. In this paper we concentrate on the extension of the tokens. The typed (colored) tokens in SNs, which can carry data, but not make full use of it, will be extended into object-oriented tokens. These can contain more subtle data and rules, for a more active participation in the control structure of the net.

INTRODUCTION

In this paper we set up the guidelines for the further development of our research in Petri Net (PN) based simulation. For the moment this is in its initial state. Our existing simulation tools can not handle the new extensions presented here.

In the following we assume some fundamental knowledge about PNs.

Simulation Nets - A brief overview

A SN uses most of the classical PN extensions like inhibitor arcs, time, colored tokens etc. Some new extensions are added, such as interrupt arcs, stochastic or-logic and hierarchies of nets. What might be characteristic for SN is also the way of modifying the classical extensions. The modifications are done for simplifying simulation modelling.

SN have been a research interest at the department since 1981 (Törn 1981) Two tools have been developed. SimNet, which is written in Simula and available in VMS and MS-DOS, has automatic statistics collection, animation (MS-DOS) and most of the SN properties described above. XSimNet is a further development of SimNet and was the first tool in our research to include colored tokens and textual extensions (code) in the graph. The colored tokens were however implemented in a restricted way. They can have data attributes and are able to store data, computed in transitions. These attributes can however not influence the execution of the nets. The experience with XSimNet have convinced us that it would be worthwhile to increase the modeling power of SNs. This includes also transitions, places and the subnet concept, but this paper will only cover the extension of the tokens.

Problems in PN modelling

The qualities of PNs in modeling of systems have been documented in several papers and research reports. When the model is simple and small the graph is also very readable, providing an easy understanding of the problem. With growing complexity this clarity might be lost. There may be so many constructions of more or less trivial primitives in the net that the overview is lost. Looking at existing PN extensions, most of them contain textual information in addition to graphical symbols. The reason is quite clear. Textual code with a well defined syntax can increase modeling power and may reduce the size of the graphical model. However, if the code is too extensive the balance between the graph and the code may be disturbed. By letting the textual declarative part grow at the expense of the graphical part, the graph could be more or less completely replaced by code in a suitable programming language.

What is a good balance between the graph and text extension? One answer to this difficult question could be: The graph should reflect the model of reality and our intensions with the model, but not contain a lot of low level constructions.

Another aspect of PN modelling problems can be stated: *One common criticism is that the Petri Net remains a static, global control structure. There is no sense in which the net consists of a collection of dynamically created objects, each encapsulating its own data and control structures, together with the ability to generate new instances of such objects* (Lakos 1994). Adoption of object-oriented structuring into PNs has resulted in the formalism called object PNs (or OP-nets). Our intension is not to apply object-oriented paradigms in the SN. The net should remain as the traditional global and static control structure, and as a complement, the tokens will be extended from passive data tokens to dynamic and data-oriented control structures. The well defined paradigm of object-oriented programming is then applicable.

Being the dynamic part of the net, the tokens are natural candidates to carry even complex code. A lot of low level details can easily be encapsulated in an object-oriented token.

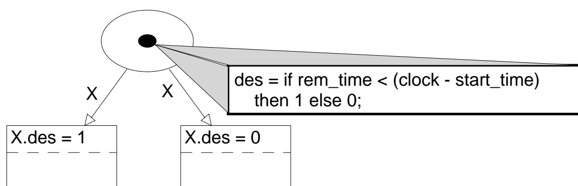
OBJECT-ORIENTED TOKENS

Introducing object-oriented properties in tokens

The step from colored tokens to object-oriented tokens is very straightforward. The declaration of a token, with type or color, is like a class and the tokens are the objects. We list the traditional object-oriented facilities and analyze how to handle them.

- Methods are an ordinary extension of the existing data attributes. They will only operate on the local data of the token object or the arguments. When a token resides in a place or transition, it could possibly also use the local data of the corresponding node. Global attributes are read-only.
- Inheritance must be implemented so that one token type is a subtype of another. Multiple inheritance will not be implemented, at least not in the first stage and a token cannot include another token type as a field. One strange thing is of course the appearance of virtual tokens, i.e. token types never intended to appear in the net. They act only as superclasses for other tokens. There is however one useful effect of this to be used in the net: If an arc is restricted to transfer a certain token type, it is valid also for all possible subtypes.
- Data hiding can be applied directly without modifications.

With respect to simulation, these extended tokens can participate in a model in a more sophisticated way. Even in a colored Petri net, the behavior of the model is dependent on the net architecture. If we add the possibility that a token may change it's attribute values, it can also influence the routing of itself in the net. As an example we can have two transitions with enabling predicates that are mutually exclusive. In these predicates both will call a certain method of the token to be used. Based on its knowledge, the token will make a choice between the transitions by returning the corresponding value from the called method. For an example see below.

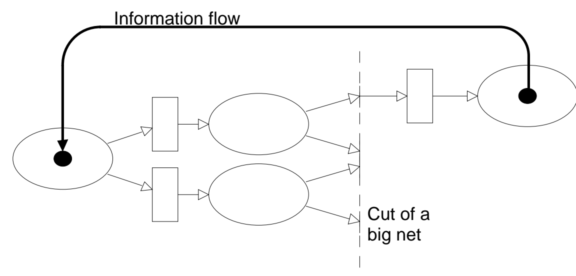


In simulation modeling we can let object tokens represent dynamic entities like cars, trucks, boats, messages etc. The net will model the static environment in which these entities occur. This has also been used before, but the object tokens can handle more of the decisions and the modeling conditions. In reality we will find many situations corresponding to such a model. A truck driver for example, can make a decision of his routing, but the choice is restricted by the statical environment, i.e. the road system. If he is hungry, he can decide to stay longer in a certain place (state), but if he is out of time he has to continue. What we see is an interaction between a static environment and complex dynamic objects.

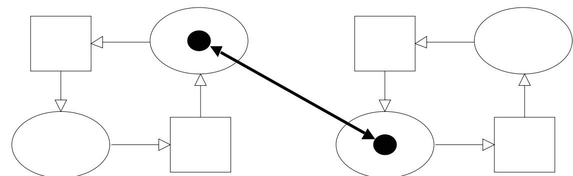
AI in tokens and token communication

A colorless token is anonymous, and as a consequence it is historyless. No matter in which state it is in the net, the next step is independent of it's lifetime up to now. If the token has identity, data and in particular methods, it can remember it's history. It can make decisions based on knowledge and it can learn by experience i.e. the basis for an intelligent choice is established.

In order to extend the knowledge base and decision making of tokens we may let them communicate with each other. This means that one token, based on it's own experience, can send information to another. There can be a communication between tokens which are in the same net (environment), but in different states and have different experience. If a token have traversed the net along a specific path it can communicate with tokens to enter. The message might be “Don't take this road” and/or a description of the path.



We can also have a model with two different nets (worlds, systems). The tokens can then transfer information between them.



The implementation of the communication between tokens can be done in two ways.

- A common data pool is the simplest alternative. According to what is common in objectoriented style, tokens of a certain class can have some variables in only one, shared instance. This restricts the communication to reside only between the same type or subtype of tokens.
- A channel, where the messages are transferred, is a more sophisticated concept. It includes the possibility of explicit addressing of a specific token. To achieve most efficiency, there must be several ways to find and identify the tokens.
  - All tokens have a unique id number, given at creation, as a standard attribute. If a token supervises others, it can store their id numbers in a list. The number can also be asked for.
  - Tokens in a certain place can be found according to the place name. Since this tokens always are queued, there must be a specific keyword for finding the next element.
  - Tokens which are not created by transitions in run

time, but by the designer, can be given a global unique name, from which they also can be accessed.

Token duplication

Tokens are traditionally dynamical objects. They are created and destroyed by transitions. Up to now there have always been a transition involved, when tokens are to be deleted or created. Another, and in some situations better, solution is to permit duplication of tokens, according to specific rules. We will than remove the privilege of token manipulations from the transitions and permit tokens to take part in the process themselves.

We will present two major strategies.

- Token-forking. A token can split up into two parts, identical up to the id number. The process is equivalent with a biological cell division. It is also very similar to process forking in the UNIX operative system.
- Supervised token duplication. One token is responsible for creation of other tokens, which also can be of different type. The creating token may even have the privilege to put the other tokens in any place in the net. In order to manipulate the new tokens, the creator must save their id's.

Analogue to creation, destruction of itself or some other token should be possible.

An example from practice: A supermarket might have a row of cash-desks. The number of cash-desks is fixed, but some of them might be closed if there are few customers. At a certain time, when more customers arrive, the number of cashiers at work will increase.

A model of this example might be SN with a place, cashes, that contains tokens of a type 'cashier'. The number of tokens in that place should increase and decrease dynamically, according to the queue length of one or more other places, representing customers waiting for payment.

Token visibility

One way of introducing time in PNs is to equip tokens with a so called time stamp (Jensen 1992). As a consequence the token is kept invisible in a place for a certain time, and cannot be consumed by a transition.

An extension of this invisibility is that it must not always be connected to a time stamp. An intelligent token should have the possibility to set itself invisible explicitly for other reasons, like simply refusing to be consumed by a transition. Such situation might be when the token is waiting for a message from another token, possibly somewhere else. The visibility setting can also in some situations be a softer way of manipulation than repeated creation/destruction.

Language for the methods

For programming object-oriented tokens in SN, we must define the language and in particular it's level. We want a very high level and expressive language with adjustment to the SN environment. Some of the desired properties could be summarized as follows:

- As there is a development towards AI, the code must be related to the problem and also permit symbolic computation. It should have no or at least few type declarations, type inference is preferred. Pattern matching in a prolog like style is an interesting feature. The language should be able to handle lists and similar complex structures.
- Automatic invoking is a usable idea. It can be defined in different ways.
  - One possibility is to put methods in time queues. One can introduce *signal(T,M)* as a procedure for automatic invoking of a token method *M* at a specific time *T*.
  - If a token has a method with the same name as a place, this method is invoked when the token arrives at the place.
  - If a token has a method with the same name as itself (=class name), this will be invoked when the token is created.
- Fuzzy logic in conditions, complementing traditional boolean values.

Most languages are designed to be stand alone. In this case we don't have a 'main program' or 'program start'. The main flow is expressed by the net and the coded algoritms are invoked on special conditions.

ACCESSING OTHER PARTS OF THE NET

To achieve full effect of the token control, tokens must be able to inspect different entities in the net. There will be a need to at least read attributes in the nodes.

In a SN transitions are equipped with code, which is executed when the transition fires. Being the active part of the net transitions can communicate with tokens directly by calling its public attributes. Therefore there is probably no need for introducing the possibility for calling transition methods from outside.

Places ar not active in the meaning that they can change themselves or in particular anything else. They do not, as the active transition, execute some code. They have however standard attributes (queue discipline, number of tokens etc.) and could be equipped with methods for accessing these attributes.

EXTENDING EXISTING CONCEPT

A token in a SN is always of a specified type. There is a possibility to declare user defined types. This is done by giving a name to the type, set its priority and list the attributes. The attributes have so far all been numerical.

Even if no types were declared by the user, there is one predefined type *any*. An *any* token can be regarded as a simple black and white token and is meant to be used if there is no need for identity and data. It has no data attributes, only a predefined priority and id.

The keyword *any* can also be used in an arc expression, which restricts what token types should be accepted by a transition. But in this case it does not restrict the desired token to be of type *any*, it rather means that all token types are acceptable.

These two concepts, which can seem to be contradictory to each other, are reconciled by introducing *any* as a

superclass for all other token types. It will than be equipped with four attributes, considered to be necessary for all tokens.

- *iflag*, invisibility flag, is a hidden boolean variable.
- *prior*, priority for queueing, is a public numerical variable.
- *id* is a public read-only numerical variable, unique for all tokens and given at creation.
- *time(T)*, is a public method for setting invisibility for a time period *T*.

CASE STUDY: A GAS STATION PROBLEM

Problem Definition

The original of this problem is found in (Schriber 1974). Some smaller modifications have been made for increasing its usefulness in demonstrating the features of the extended nets presented.

The interarrival times of cars approaching a gas station with the intention of possibly stopping for service are distributed as shown in the table below:

Interarrival time (seconds)	Cumulative Frequency
< 0	0.00
< 100	0.25
< 200	0.48
< 300	0.69
< 400	0.81
< 500	0.90
< 600	1.00

A car stops for service only if the number of cars already waiting for service is less than or equal to the number of cars already being served. (That is, a car stops only if the driver perceives that not more than one car per attendant is already waiting to be served.) Cars which do not stop go to another gas station, and therefore represent lost business. Other data:

- Opening time: 7 - 19. Cars already waiting in line at 19 o'clock are served, but cars arriving later than that are not accepted for service.
- Average of the profit per car served: $1.
- Salary of attendants: $2.5 per hour, payed for 12 hours a day.
- Fixed costs: $75 per day.
- Service time distribution:

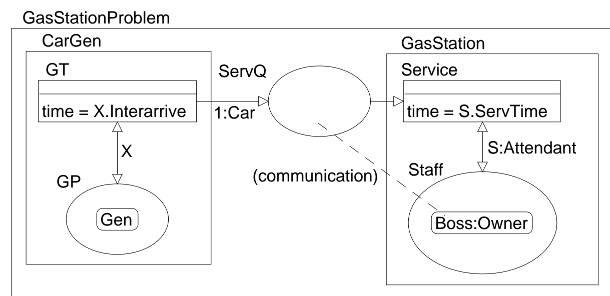
Service time (seconds)	Cumulative Frequency
< 100	0.00
< 200	0.06
< 300	0.21
< 400	0.48
< 500	0.77
< 600	0.93
< 700	1.00

The station's owner wants to determine how many attendants he should hire to maximize his daily profit. We will add one feature, that the owner also works as a attendant himself. The goal will be to build an intelligent SN which via rules finds the optimal number of attendants.

Solution

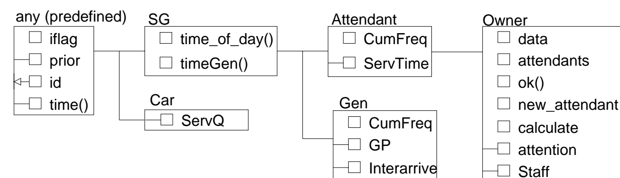
We will introduce a token type *Owner*, which exists from the start in one single instance. This gives us the possibility to assign it a global name, *Boss*, permitting other tokens to address its methods. At the beginning of the simulation the *Boss* works alone. "He" will collect data all the time and after some days an economical analysis is done. Another attendant will be hired if the lost profit is larger than the salary of an attendant.

Because of the possibilities of token control, the SN is very simple and the high level model is a clear representation of the problem.



Cars are generated by *CarGen* and deposited i *ServQ*. After arrival they send a message to *Boss*. If the answer is no (=to many cars waiting), the *Car* token deletes itself, symbolizing left off for another station. The communication allows *Boss* to maintain the control of incoming cars.

The *Gen*-type token is responsible for the timing of incoming *Cars*. Every time it visits the place *GP* there is a corresponding method invoked which controls the time of day. If it is over 19 o'clock, *Gen* will set itself invisible until next morning. Visiting transition *GT* its method *Interarrive* returns a time interval calculated according to the above table. An *Attendant* token has a similar method, *ServTime*, for the transition *Service*. This gives the following structure of the tokens.



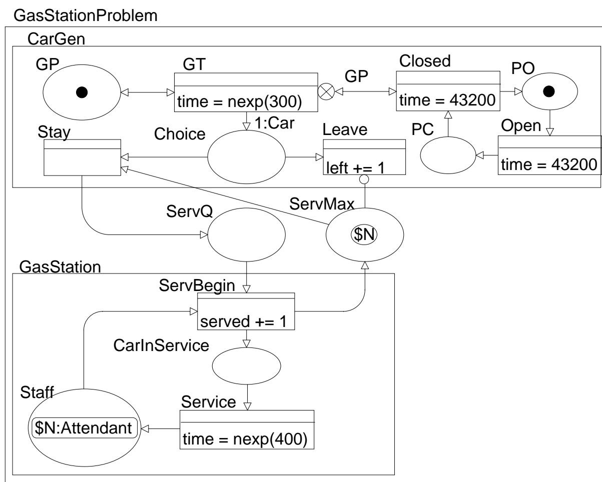
The picture shows how inheritance is used, with the predefined *any* as a superclass for all. *Owner* can work as an *Attendant* himself (= be accepted of *Service*) because it is inheriting from the *Attendant* class. *SG* is never used as a real token, it is just a superclass containing what *Attendant* and *Gen* have in common i.e. a function for translating the total time in seconds to day time and a time interval generation function for different parameters.

According to the details above the behavior in the net can be the following. The method *ServQ* in *Car* is invoked when arriving in place *ServQ*. The *Car* calls *Boss.attention*, and will immediately know if it shall join the queue or not. The method *Staff* in *Owner*, will give *Boss* an opportunity to check the time after each job. When closing time *calculate* will be called to sum up the day's result. Each tenth day, for example, *Boss* can make a decision if he should hire another *Attendant*, e.g. create a new token. The model can easily be extended and made more complicated in different ways:

- *Attendant*s can have lunch-times, all at different times.
- *Boss* can also delete *Attendant*s or they can decide to leave the job.
- The requirement on the queue length for the *Car*s to stop can be individual.

Alternative model in SN

Let us se how to model the *Gas Station* problem in a SN, as it is today.



In this model there is no possibility to self adjustment, performed by intelligent tokens. We must do several simulations with different values of variable $N. According to output, we must calculate the optimal solution. There are two transition attributes, *left* and *served*, for checking lost and earned profit. The control structure is fixed in the net and hard to extend. In general this model is much simpler and weaker.

The time distribution must be approximated with exponential times, because SNs do not have possibilities for deeper programing.

The model uses one significant extension of SNs, an interrupt arc between *GT* and *Closed*. Using traditional inhibitor arcs the model would have been even larger.

CONCLUSION AND FUTURE PLANS

Tokens extended in the way described above, seems to be an effective way to decrease the size of the net. It is an important facility, but not the only reason to introduce them. The advantages could be summarized as below.

- Division of the static and dynamic areas of the model in separate parts, represented by the net and the tokens. This allows more natural modelling.
- Introduction of AI (rules and data) in the model, providing for finding of the optimal solution automatically.
- The textual code is connectd to the graph in a structured way.

The goal of our research is the construction of a simulation tool, based on experiences from XSimNet and the theoretical work on extension of the net. The concept of object oriented tokens is only one part of this project.

REFERENCES

Evans J. B. 1993. "The Net Semantics of Demian." Technical report TR-93-11. Department of Computer Science, Faculty of Engineering, University of Hongkong. (Dec.)

Ghezzi, C., D. Mandrioli, S. Morasca and M. Pezz'e. 1991. "A Unified High-Level Petri Net Formalism for Time-Critical Systems". IEEE Trans. Software Engineering (Feb): 160-172.

Javor A. 1993. "Petri Nets in Simulation". EUROSIM - Simulation News Europe. no. 9 (Nov): 6-7.

Jensen, K. 1992. Coloured Petri Nets. Volume 1. Spinger-Verlag, Berlin, Heidelberg.

Lakos C. A. 1994. "LOOPN++: A New Language for Object Oriented Petri Nets." Technical report TR94-4. Computer Science Department, University of Tasmania.

Papelis Y. E., and T. L. Casavant. 1992. "Specification and Analysis of Parallel/Distributed Software and Systems by Petri Nets with Transition Enabling Functions". IEEE Trans. Software Engineering (Mar): 252-261

Peterson, J. 1981. Petri Net Theory and the Modeling of Systems. Englewood Cliffs, N.J. 07632: Prentice-Hall.

Reisig W. 1985. Petri Nets, An Introduction. Springer-Verlag. Berlin Heidelberg.

Schriber T.J. 1974. Simulation using GPSS. Whiley, New York.

Törn, A. 1981. "Simulation Graphs: a general tool for modelling simulation designs". SIMULATION 37:6: 187 - 194.

Törn, A. 1991. Simulation Modelling. Åbo Akademi University, Reports on Computer Science & Mathematics, Ser. B, No 12, 140 pp.

IWong C.Y., T. S. Dillon and A. E. Forward. 1985. "Timed Places Petri Nets with Stochastic Representation of Place Time." In proceedings of International Workshop on Timed Petri Nets (Torino,, Italy, July 1-3). IEEE, Computer Society Press, Silver Spring, 96-103.