Proceedings of the Conference on Modelling and Simulation 1995 (ESM95, Prag)
eds. M. Snorek, M. Sujansky and A. Verbraeck

DECOMPOSING SIMULATION NETS BY TOKEN COMMUNICATION

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

ABSTRACT

The usual way to handle larger problems is to divide them into subproblems. Traditionally this is a weak point in Petri nets. Introduction of hierarchies in the net is one solution to the problem. An alternative way to decompose Petri nets is to model every subproblem as an independent net. These nets are linked together by communication between programmable object oriented tokens.

INTRODUCTION AND BACKGROUND

Simulation Nets (SNs) are Petri Nets extended for convenient modelling of discrete event simulation problems (Peterson 1981, Reisig 1985, Törn 1981, Törn 1991). Two tools have been developed for execution of SN models (Gustavson and Törn 1994a). Object Oriented (OO) Tokens, for increasing the modelling power of SNs, have been introduced in (Gustavson and Törn 1994b). Tokens are natural candidates to carry even complex code and the step from coloured tokens to object oriented tokens is very straightforward. If a token has identity, data and in particular methods it can remember its history, make decisions based on knowledge and 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 let them communicate with each other. This is the most significant difference between our token model and other similar models (Javor 1994, Lakos 1994).

These programmable tokens with intercommunication facilities are very powerful in modelling. By their properties they become the dynamic part of the otherwise global and static control structure of the net. They can be used for adding continuous and fuzzy properties to the net. In the following we will focus on another quite interesting consequence, namely modularisation.

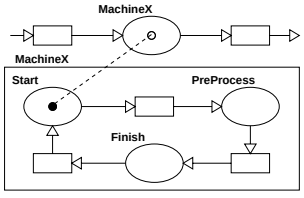
DECOMPOSITIONS OF NETS

The absence of compositionality has been one of the main critiques raised against Petri net models (Jensen1992). One immediate answer to this critique has been the construction of hierarchical Petri nets. SNs are well suited for structuring a model as a hierarchy of submodels. The structure of the composition is a tree with one root net containing all the subnets.

As a complement to the hierarchical composition we introduce a division into components which are unordered. For this purpose we use the token communication described above. We split the model into several nets and have the tokens to transfer information between them i.e. the only link between the nets is the token communication. Let us name this unordered modularisation aspect decomposition.

To make the idea simpler from a theoretical point of view we let the details of the token communication be hidden and considered as an implementation problem. For the moment we can simply think of several different nets and one and the same token appearing in all of them at the same time. Occurrences of the token in one net will influence on the behavior of the other net. Nevertheless the nets are unrelated in the meaning that they don't need to "know about each other", i.e. each net expresses its own aspect of the problem. This approach lets us concentrate on one aspect at a time when modelling and gives as result a net of that aspect. Depending on the states of a common token, which is the glue to other aspects (modules), there are several outcomes which are not obvious from the graph. However, when modelling the aspect we don't have to care about how these states occur or change.

A typical simulation example, a production line model, may be used to clarify the idea. Assume that tokens (items) pass through different machines in sequence. In a high level model the process of a token in one machine can be represented by a place (*MachineX*), where the token stays for some time. Suppose that we want more complexity, so the phases in the machine processing should be modelled in detail. Rather than inserting a subnet at the location of the place we model the machine in a separate net. When a token arrives to the place, its counterpart in the machine net starts the inner process and the upper level token becomes available again after the machine process has finished.



ASPECT DECOMPOSITION VERSUS HIERARCHICAL

The example of a production line is a model that can be structured in a hierarchical way as well. The rules of a Petri net permits every place or transition to be substituted by a

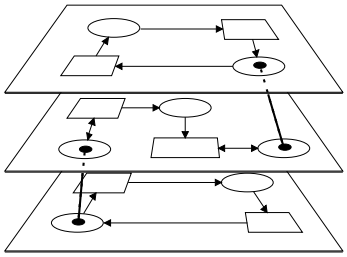
net. Instead of the place representing the machine we insert the machine net. The question of the advantage of aspect decomposition is then raised.

We claim that there is a different philosophy behind the aspect decomposition. It has to do with dependency. The intention is that the net modules which appears in an aspect decomposition should be complete nets. They should be executable on their own. Of course the coupling to the other parts must then be replaced by assumptions. In our example the machine net can be executed stand alone if we assume that the incoming trigging signal has arrived. In that meaning we set up some preconditions. It is however not clear that the execution of only one part is meaningful in the sense of the semantics of the simulation problem expressed by the complete net.

This requirement of completeness of the module should be applicable to analysis. The different net modules should be independently analysable, provided the preconditions are given. In that sense the correctness analysis of the different modules can be carried out one by one. We will not here go into details about analysis of SNs, which are high level Petri nets and to some parts hard to analyse. The work with analysis of SNs is yet to be finished. The intention is that the traditional Petri net analysis technics (invariants, reachability etc.) should be applicable to the modules.

For the correctness of the complete model one has also to consider the connections between the modules. The outgoing messages from one net can then be regarded as postconditions, which are transformed into preconditions in another module. Since these signals are fairly simple, e.g. variable assignments, their correctness is easy to overview.

Hierarchical decomposition is more a programming construction than a true modularisation. A hierarchically decomposed net can be flattened and its decomposition is really the concept of data hiding. It is principally a sequential way of thinking. Aspect decomposition associate to the parallel paradigm and the coupling is indeed a synchronization, despite the fact that data exchange may be involved. One might think of nets in different dimensions, where one level throws its shadow on another level but is independent otherwise.



In addition we want to mention some pure practical modelling properties of aspect decomposition.

- It can always replace a hierarchical modularisation and will therefore always be a complement. The choice is done by the programmer.
- If a subnet occurs several times in a net at different locations it must be reproduced in each spot. One aspect module is enough to replace them all.
- Depending on the model the aspect decomposition can be clearly preferable. In some models, like the example of the ferry example below, where modules with a high degree of independency can be found in a natural way, the hierarchical decomposition is hard to apply.

IMPLEMENTATION DETAILS

In reality all tokens must be different objects. They cannot be in more than one place at a time. How can we than express the simplified model above, where one token can occur in more than one place? The concept of one token in two nets can easily be expressed in terms of communication between OO tokens. The object oriented paradigm is also included in the possibilities.

OO Tokens, short overview

The major intention by introducing OO tokens was their extended role in the model, e.g. introduction of intelligent simulation in SNs. This consists of an automatization of an otherwise iterative interaction between the programmer and the tool: The result of the simulation is analysed, the model is corrected and simulated again. By equipping a model with knowledge and rules for the goals of the simulation, the bottlenecks could be discovered and removed at run time.

In our model this work is delegated to the tokens. We do not permit dynamical changes in the net structure. In this case we cling to the paradigm of Petri nets, where the net is fixed and tokens are dynamical and can be destroyed and created. For this purpose OO tokens are equipped with considerable computational power. The following properties are of interest:

- Communication between tokens are introduced in two ways.
  1. The possibility of explicit addressing of a specific token can be done with several ways to find and identify the tokens. We can use the unique id number, given to all tokens at creation as a standard attribute. Tokens which are not created by transitions at run time, but by the designer, can be given a global unique name, by which they also can be accessed.
  2. Tokens of a certain class can have some variables in only one, shared instance according to what is common in object oriented style. This common data pool restricts the communication to reside only between the same type or subtype of tokens.
- Tokens are permitted to duplicate or destroy themselves, according to two major strategies:
  1. Token-forking. A token can split up into two, identical up to the id number. The process is equivalent to a biological cell division. It is also very similar to process forking in the UNIX operative system.
  2. 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 created token in a place of its own choice.

❍ An intelligent token has the possibility to set itself invisible. As a consequence a token in a place cannot be consumed by a transition. This is a way of introducing time in places but can also be used for other purposes, e.g. having a token to wait for a message from another token, possibly residing somewhere else.

OO Tokens in decomposition

All these properties can be used in decomposition. The strategy chosen depends on the model. In the example of the product line we can use the visibility concept. The detailed machine net is designed with a start up place, initialized with an invisible token. When a token in the production line net arrives to the place corresponding to the specific machine, it sets itself invisible and sends a message to trig the machine. When the machine token gets the signal it sets itself visible. It will then immediately be consumed by a transition which starts the inner process. After finishing it sends a reply so that the upper process can continue.

However, closer to the idea of one token in several nets is to use shared memory for tokens in different nets. As soon as one token is updated the others will know about it. How the changes in their memory occur can be hidden from them. They just follow the state of their inner data without asking. As a consequence the modules become more independent.

Because of the different roles of the tokens in different nets they can be declared as different classes in the object oriented meaning. The requirement is that they have a common superclass from which they inherit the shared data. It follows that the different tokens can have completely different methods.

Simulation example: Car Ferry

We will use another simulation example, that of car ferry, which is inspired by (Birtwistle 1979). It will better show how the description of one object can be completely decomposed according to different aspects although the events in these descriptions occur simultaneously. To make it more interesting we refine the model. The original model concerns only the aspect of transporting cars. This will be enlarged to include also weather conditions and the inner state of the ferry. We will regard these as three aspects:

① A ferry serves motorists wishing to cross the strait between the mainland and a small island. It is the original model with the following data.
- Capacity limit is six cars.
- Working time 07.00 - 22.00, Time is checked before leaving mainland. It will not leave after 21.45.
- Cars have some interarrival times and the crossing time is normally distributed.
- The wanted data is the average number of cars/trip, empty crossings, average waiting time etc.

② The ferry has inner states which will influence on its behaviour.
- It has a limited amount of fuel, which is decreased for each trip.
- Its functioning is subject to impairment.
- Errors in machinery might occur randomly using some reliability model.

③ The weather conditions are changing and will influence on the crossing time and the condition of the ferry.
- The season is changing, causing ice or storm conditions to change.
- The wind has several directions (states) and power.
- The occurrence of fog and rain will influence on sight conditions.

When modelling we construct three SNs according to this scheme. We will have a certain token representing the ferry. The ferry occurs in the following structure of the three nets:

① A model containing the environment with the strait, car queues and generation of cars. The ferry moves between two places representing mainland and island, loading and unloading cars.

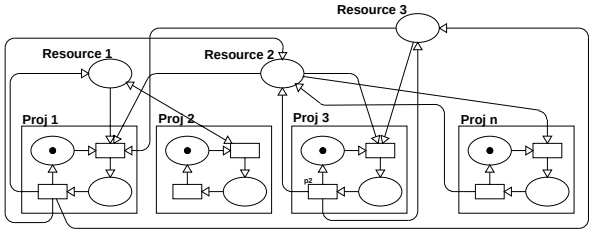
② This net reflects the inner condition and will be a reliability model (MTTR factor). Significant states as `out of fuel' and `need for repair' can be represented by places into which the ferry moves.

③ The weather environment is modelled as different states of seasons, wind direction etc. In this net we can have more than one instance of the ferry representing all weather aspects.

In the original model the crossing time is just the normal distribution with some mean and deviation. In the enhanced model the ferry will compute the speed before crossing and then the required time can be derived. The speed is calculated from shared state variables in the token, which are updated in net two and three.

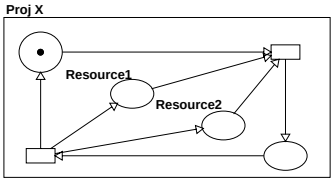
DISTRIBUTING GLOBAL RESOURCES

Typical for some simulation models is the use of a global resource, expressed by a token in a place and shared by different parts (transitions or subnets) of the net. A model can have more than one of such resources. The complexity of the graph is increased if different parts of the net need global resources. No plane embedding can be found and very long and complicated arcs have to be drawn. In case of hierarchical decomposition the resource places has to be put at the uppermost level, violating the structural construction of the net.



How can we apply aspect decomposition to models with shared global resources? This subproblem can be solved by the same technique which made aspect decomposition pos-

sible, namely that the same token can appear in several places at the same time. Making use of that we let each part of the net which needs a resource get its own copy.



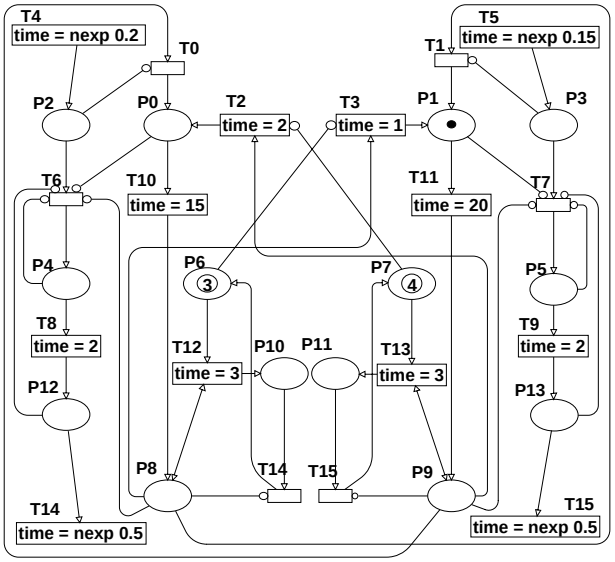
Since all these copies model the same resource only one token can be consumed at a time. The mechanism which permits such a construction is token communication, in the same manner as in the decomposition. When one token is consumed the others are locked in their places by the inner construction of OO tokens. The distributed places can occur in different modules on different levels in the net. The effect is the same as having only one place.

EXAMPLE: TRAFFIC LIGHT

We will introduce a detailed example which is taken from a real simulation research concerning traffic lights. A complicated model in a Petri net will be decomposed into three rather simple SNs.

The Original Petri Net Model

This model below is in the form it was used for experimenting with in real traffic situations. It models a crossing with roads going in two directions, north-south (N-S) and east-west (E-W).



To understand this net one has to consider that transitions are equipped with time delayed firings and by consequence priority for the immediate transitions is assumed.

Direction E-W is modelled in the following way:

- ❍ Generation of cars: T5.
- ❍ Queue for crossing: P3.
- ❍ Departure is modelled by T7-P5-T9-P13-T15. Two transitions are required for modelling an exponential time following a fixed time.

Direction N-S is modelled by T4-P2-T6-P4-T8-P12-T14.

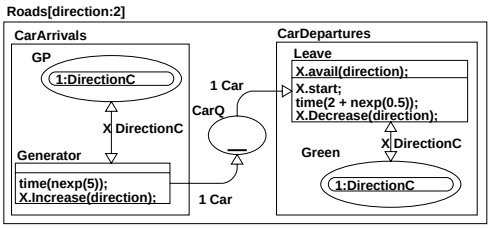
The Light is controlled by a token in P1, which means green for N-S. Transferring of the token to P0 means green for E-W. The complicated control structure can be described by an algoritm, where the real world events are given in comments.

```
p1                      ;direction N-S is open
wait 20                 ;fixed minimal time
t11->p9
for i = 1 to 4          ;repeat 4 times(
   if (p2 is empty)     ;if no cars in N-S
      then t0->p0       ;close N-S, open E-W
   else wait 3          ;additional time)
if (p2 is empty)        ;if no cars in N-S
   then t0->p0          ;close N-S, open E-W
wait 2                  ;additional time
t2->p0                  ;close N-S, open E-W
```

Decompositional Model

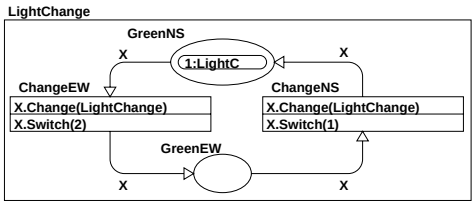
The result of a decomposition in SN is three separated nets. Because of the realistic programming in SN it contains a lot of details. We will not comment all of these but concentrate on the most interesting parts.

① The aspect of the cars crossing is modelled by a generator subnet, a queue place and a consuming (departure) part. For easier understanding we will point out that this part of the net is modelled by time consuming transitions. The notation in the name, [direction:2], means that this is two identical nets, separated by the local variable *direction.* The values 1 and 2 for *direction* corresponds to N-S and E-W.



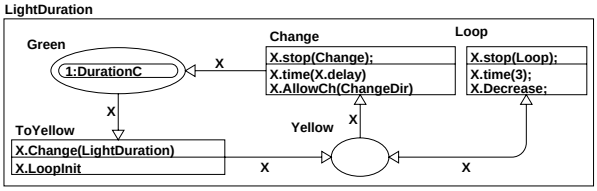
The model follows the classical *single server single queue* concept. The generation part works all the time, but the departure stops when the token in place *Green* sets itself unavailable. On this level we do not bother why and when this occurs.

② The control of the lights is very simple on the uppermost level. The token of type *LightC* will stay in one place for a time, modelling green for a certain direction.



The actual time is computed somewhere else, so the details are hidden. After the time has passed it will switch to the other direction.

③ The complicated part of the control is modelled as a net of its own. The same net performs the time delay for both directions

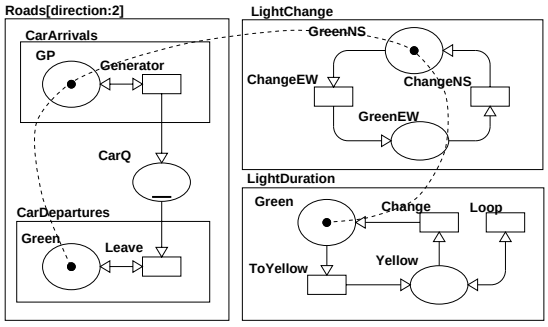


The light duration algoritm described above can be expressed in the following way.

❶ The token is waiting in place *Green* for a minimal time.

❷ The token moves to place *Yellow*. In case of no cars in queue (in the *Roads* net) it moves back to *Green* immediately. Otherwise it goes into a delayed self loop, modelled by the transition *Loop*.

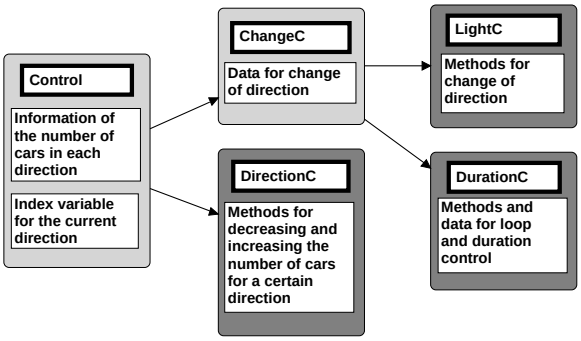
On the uppermost level we shall consider the connection between the modules. In the following picture the dotted line express the relationship between the instances of the same control token.



The token is involved in both creation and destruction of cars. By updating data after both operations it will always know how many cars are queued up in each direction. Because of its appearance in the *LightChange* net it knows which direction is open. When modelling the duration of time it can always deduce when to change or when to loop as it knows if the queue is empty.

Finally we shall look at the implementation of this token. As can be seen the tokens in the models are not even of the same type. This is however well motivated, because even if they work on common data, they perform different operations on the data and appear in different environments. The object oriented concept gives a very natural shape to the structural differences between these tokens.

The light grey tokens in the tree structure below are virtual tokens acting as superclasses. The others really appear in the net, as can be seen form the graphs above. The *Control* token consists only of shared data areas to store the information of which direction has green and the car queues. It is the main communication data base. The type *DirectionC*, involved in the *Roads* nets managing the passing of cars, inherits directly from *Control*. Only this token type can update the number of cars. The types *LightC* in the net *LightChange* and *DurationC* in *LightDuration* needs some common data for the change control which is of no interest for *DirectionC*. They inherit this data from the type *ChangeC*, placed between them and *Control*.



Tokens of type *Car*, which are generated and consumed, contain no data and therefore need no detailed description in this model.

REFERENCES

Birtwistle G.M., 1979, “Discrete Event Modelling on Simula”,(Macmillan computer Science series) 1. Computer Simulation 2. DEMOS,p 67.

Gustavson Å. and A. Törn. 1994a. "XSimNet, a Tool i C++ for executing Simulation Nets." In *Proceedings of 1994 European Simulation Multiconference* (Barcelona, Spain, June 1-3 1994), 146 - 150.

Gustavson Å. and A. Törn. 1994b, “Object-Oriented Tokens, A Way of Increasing the Modeling Power of Simulation Nets”, In *Proceedings of 1994 European Simulation Symposium* (Istanbul, Turkey, Oct. 9 - 12 1994), 97-101.

Javor A. 1994. “Intelligent Objects in Simulation models”, In *Proceedings of 1994 European Simulation Symposium* (Istanbul, Turkey, Oct. 9 - 12 1994), 9 - 13..

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.

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.

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.