CAUSAL-SEMANTIC-BASED SIMULATION AND VALIDATION OF HIGH-LEVEL PETRI NETS

Jörg Desel, Thomas Freytag
Institut AIFB, University of Karlsruhe, Germany
E-mail: {desel | freytag}@aifb.uni-karlsruhe.de

Andreas Oberweis
Lehrstuhl für Wirtschaftsinformatik II, University of Frankfurt/Main, Germany
E-mail: oberweis@wiwi.uni-frankfurt.de

KEYWORDS

Petri nets, causal semantics, simulation, processes, validation of properties

ABSTRACT

This contribution describes a simulation concept for systems modelled by high level Petri nets that is based on causal semantics. Dynamic properties can be checked by evaluating these partially-ordered simulation runs. It will be shown that this approach is not only more efficient but also has more analytical power than the "classical" simulation method based on sets of totally-ordered transition occurrences.

INTRODUCTION

Petri Nets have become a widely accepted formalism for modeling, simulation and analysis of complex systems in a variety of application domains. In particular high-level Petri nets are applied because of their flexible and compact structure.

There are two general views at the dynamic behaviour of a net model: The first one - called the *sequential semantics* - looks at the set of *occurrence sequences* of a net. The second one - called the *causal semantics* - looks at the set of *partially-ordered runs* (or *processes*) of a net. Whereas the causal semantics is favorized in Petri net theory (e.g. [Rei86]) because of its ability to handle concurrency, the area of applications is dominated by sequential semantics because of their easy and straight-away definition. Usually, simulation tools for Petri nets generate totally ordered sequences of transition occurrences and thus are based on sequential semantics.

There are also two ways to check (desired or undesired) properties of a net model: The first one is to apply *analysis methods* from net theory deciding a property mostly by analysing the state space. This very clean way becomes inapplicable when systems are large and their state spaces suffer from the *state explosion*. The second approach is not to consider all possible executions of a net but only a well-chosen subset determined by *simulation*. The given property is then checked against 'empirical' data - comparable to the systematic testing of a program.

This paper sketches an attempt between these approaches - exploiting the advantages and avoiding the disadvantages of each. It will be shown that the approach of partially-ordered simulation is not only based on a well-founded formalism but also able to improve efficiency in practical applications.

The contribution is structured as follows: The following section gives a summary of the underlying formalism and provides a small example. The next section describes how partially-ordered simulation contributes to an efficient representation of the simulation data. Afterwards a rough idea is given about the simulation policy and its algorithmical aspects. Finally it will be shown that a class of properties exists that can not be checked by sequential simulation but rather requires partially-ordered simulation.

The topics of this paper are part of the work of the project "*Verification of information systems by evaluating partially-ordered Petri net runs (VIP)*" sponsored by the German Research Society (DFG). Further information can be obtained in [DO95, DFO97] or at our WWW page [DFOZ96].

BASIC NOTIONS

Predicate transition nets

We suppose the reader to have some understanding of Petri nets, in particular with high level nets as introduced e.g. in [Jen92]. For algorithmical reasons, we restrict the class of *Predicate/Transition nets (Pr/T nets)* to finite nets with finite domains and finite initial markings containing no transitions with empty preset or postset. We restrict operations on domains to take place inside guard expressions

(i.e. all arc labels consist only of multisets of variables). This is no real restriction of the net class because every net with arc label operations can be transformed to an equivalent net without them (cf. Fig. 1).

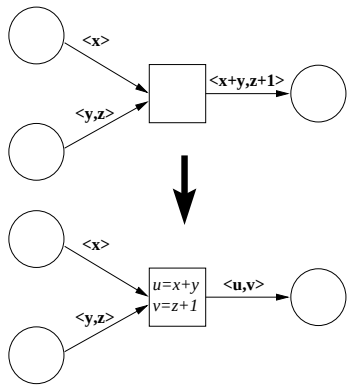

Fig. 1: Transformation of arc operations

Fig. 2 shows a small example net. It is a simplified model of an office where documents are created, checked, updated, archived and deleted. Transition **Init** shifts documents from place **Document** to place **Ready**, assigning a release number 1 to the document (**x=1**). Now there are two possible continuations: The first one is to throw a document into the wastebasket by firing transition **Delete** and moving it to place **Wastebasket**. The second possibility is to fire transition **Check** and move the document to place **Ok**. Now transition **Update** can occur putting a copy of the document to place **Archive** and passing the updated document with a release number incremented by 1 (**y=x+1**) back to place **Ready**. Initially two documents <**a**> and <**b**> are put on place **Document**, all other places are unmarked.

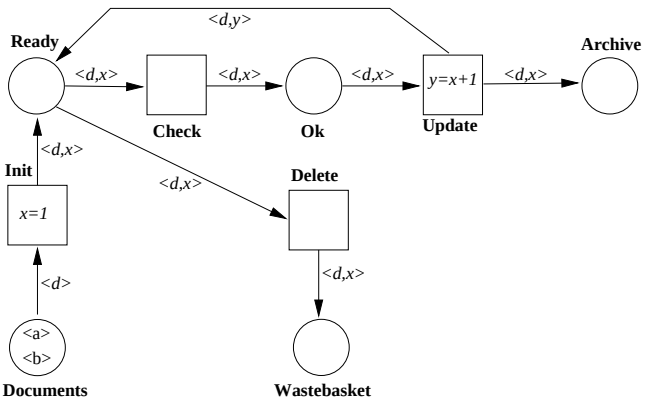

Fig. 2: An example Pr/T net

Causal nets

A Petri net is called a *causal net* if every place has at most one input transition and at most one output transition, every transition has at least one input place and at least one output place and the flow relation has no cycles (i.e. its transitive closure is a *partial order).* The places of a causal net are called *conditions,* its transitions are called *events.*

Shortly, a causal net is an acyclic, place-bordered net with forward and backward unbranched places. The places without input transitions are called *minimal elements* and the places without output transitions are called *maximal elements* of the causal net.

Processes and simulations

The dynamic behaviour of a Pr/T net is given by the set of possible executions that start at a given initial marking. Every execution of a Pr/T net is called a *process* of the net. A process can be described in a straightforward way by a causal net (then called a *process net*):

- Each condition of the causal net represents the existence of a marking tuple (multiset element) on a particular place of the Pr/T net
- Each event of the causal net represents the occurrence of a transition in the Pr/T net for a particular variable assignment
- Each arc of the causal net represents the flow of marking tuples in the Pr/T net: Whenever a transition consumes a tuple from a place, an arc is drawn from the associated condition to the associated event. And whenever a transition produces a tuple on a place, an arc is drawn from the associated event to the associated condition
- The minimal elements of the causal net are those conditions associated to the initial marking of the Pr/T net

As a notation, we label a condition of a process net by the name of the associated place followed by the associated marking tuple put in brackets. For example, a condition named **P1(a,1)** stands for the marking tuple <**a**,**1**> on place **P1**. The events of the process net are labelled by the name of the associated transition followed by the list of variable assignments put in brackets. For example **T1(x=a,y=1)** stands for the transition **T1** occurring for the assignment

x=a and **y=1**. Fig. 3 and Fig. 4 show different processes for the net in Fig. 2 (some names are abbreviated for sake of readability). The set of minimal elements in both process nets consists of the conditions **Doc(a)** and **Doc(b)** denoting the marking of place **Doc(ument)** by the tuples <**a**> and <**b**>. In Fig. 3 the event **Init(d=a,x=1)** represents the firing of transition **Init** for the variable assignments **d=a** and **x=1**. After this occurrence, place **Ready** is marked by the tuple <**a,1**> which is represented in the process net by the arc from **Init(d=a,x=1)** to **Ready(a,1)**. Note that both processes are not maximal in the sense that they can still be continued by further events (e.g. connecting **Check(d=a,x=3)** to the maximal element **Ready(a,3)** of the process net in Fig. 3).

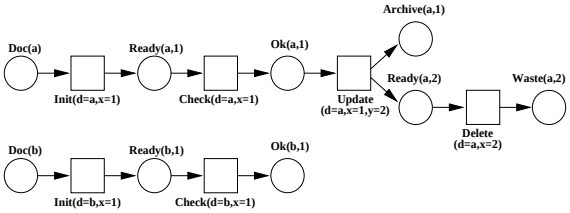

Fig. 3: A process net of the net in Fig. 2

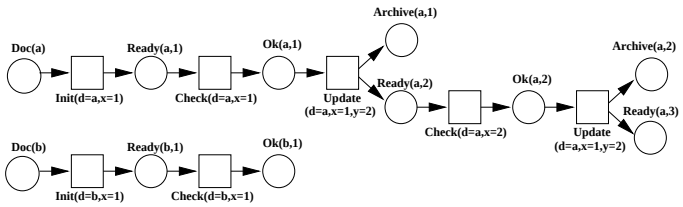

Fig. 4: Another process net of the net in Fig. 2

Causal nets cannot handle conflicts because their places do not allow branching. Conflicts are resolved by creating additional processes describing alternative continuations. In our example, there is a resolution for a conflict that arises after **Update(d=a,x=1,y=2)** has occurred. The process in Fig. 3 shows the continuation where **Delete(d=a,x=1)** occurs next, whereas Fig. 4 shows the contination where **Check(d=a,x=2)** occurs next.

EFFICIENCY IN SIMULATION DATA REPRESENTATION

The behaviour of a concurrent system is described by the set of its possible *executions*. An execution consists of a set of *events*, each of them having a certain set of *pre- and postconditions*. The *causal structure* of an execution is defined by an iterated combination of events where a postcondition of one event can be a precondition of another event. This leads to a canonical partial order of events - the *causal dependency order*. Two events are *concurrent* if and only if they are not ordered by the causal dependency order. For example it can be checked easily in the process net of Fig. 4 that the event **Update(d=a,x=1,y=2)** is a causal successor of the event **Check(d=a,x=1)**. On the other hand the events **Update(d=a,x=1,y=2)** and **Check(d=b,x=1)** occurred independently because they are not causally ordered.

A simulation based on *sequential semantics* constructs event occurrences and generates totally ordered sequences of events by forcing independent events to be ordered. The advantage of this concept is the simple and straightforward representation of a system run.

A simulation using *causal semantics* constructs event occurrences and generates partially-ordered structures (*processes*) of events preserving their causal dependency. In general, a process corresponds to many different occurrence sequences - each represented by one possible execution path of the process net.

To demonstrate the benefit of our approach we consider the example from the previous section. The process shown in Fig. 5 contains nine occurrence sequences.

2-5-1-3-4, 2-1-5-3-4, 2-1-3-5-4, 2-1-3-4-5, 1-2-5-3-4, 1-2-3-5-4, 1-2-3-4-5, 1-3-2-4-5, 1-3-4-2-5

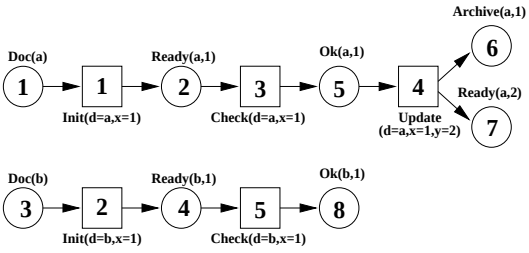

Fig. 5: A process containing nine occurrence sequences

The events and conditions are numbered merely to identify them - no sequencing is implied. It can easily be proven that the number of occurrence sequences grows exponentially in the degree of concurrency of the underlying net. So processes are a very effective data structure to store the behaviour of a system that has many different executions.

Additionally, the causal dependencies between arbitrary events are always retrievable from a process but never from a set of occurrence sequences. Considering e.g. the sequence **1-3-2-4-5** from the example above, it is impossible to decide whether the

event **3** precedes the event **4** because of a causal dependency or whether they were sequentialized arbitrarily by the simulation policy.

So simulation runs based on causal semantics have several advantages compared to the sequential approach:

- Compactness of the simulation data
- Explicit representation of the causal dependencies

EFFICIENCY IN THE SIMULATION ALGORITHM

A (partial-order-based) simulation is a subset of the set of all processes of a Petri net. One possibility for the algorithmic construction is the generation of the *complete process set* which is described in detail in [JaK89, DOZ96]. Because the set of processes can grow exponentially, this method is in most cases not applicable in practical applications. Instead, we concentrate on the construction of a well-chosen *subset* of the set of processes controlled by *termination rules* to determine the point to finish the construction of a process or of the simulation at all and by *event selection rules* to choose the next transition (and its variable assignment) to occur. It turns out that we can save storage resources in particular when simulating those systems that have long *deterministic prefix executions*.

Termination rules

Termination rules control the efficient construction of processes in different ways. *Global termination rules* determine the point when to stop the whole simulation. This could be either when all possible processes are built, or when an upper bound for the number of processes or a certain number of event occurrences is reached. *Local termination rules* are needed to stop the construction of the current process. This point could be either that no more events can occur, or a given number of event occurrences or a maximum process chain length is reached. In addition, *dynamic termination rules* can be established depending on the individual system property that is to be validated. *Finite prefix rules* as introduced for net unfoldings e.g. in [Esp94] can be adapted to processes to provide a neat criterion to determine a state where the system repeats a behaviour that has already been observed before.

Event selection rule

At any time during the simulation of the system there is a certain set of enabled transitions with associated variable assignments that are possible continuations of the system behaviour. The quality of the simulation strongly depends on the policy used to choose one (or more) of these alternative events. One such policy could be to let the user do this decision. Other policies automate this decision by doing a random choice that can be stochastically weighted e.g. with a pre-defined *priority factor*, with the number of occurrences in the past, or with a combination of both. It is also possible to let this priorities depend on the specific kind of property that is to be validated.

Deterministic prefix executions

One observable property of real-life information systems is that they may execute deterministically (conflict-free) for a certain (possibly long) time before any nondeterministic behaviour (i.e. conflicts) takes place. This turns out to be a storage-wasting procedure because all different continuations after the conflict have an identical prefix that has to be stored to represent the complete process history. This reduncancy can be avoided by an efficient structuring of the simulation data.

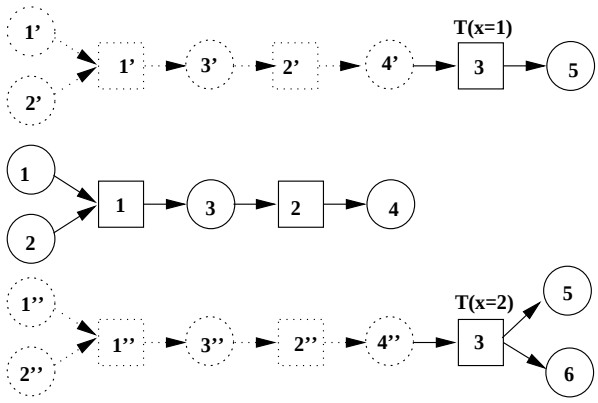

Fig. 7: A shared deterministic prefix

The system must be able to recognize these situations and store the identical data only once by "sharing" the common prefix with each process net that is a continuation of it. This principle is comparable to the use of a reachability graph when using the sequential simulation method. Two states that are reachable via different occurrence sequences but represent the same marking are mapped to the same

node. Fig. 6 shows an example of a determistic prefix that is common to two process nets (shared conditions and events are drawn in a dotted style).

EFFICIENCY IN PROPERTY VALIDATION

In opposite to the sequential semantics it is always possible for two causally ordered events to tell their "causal distance" - i.e. the number of events occurring between them.

We want to try checking the following property of the system: *'Is there an execution where a document is created and destroyed without ever being checked?'*. This property is very typical in the verification and optimization of concurrent information systems when we are searching for erroneous execution paths. Fig. 7 shows a graphical representation by adding a *query transition* to the net graph. Query transitions are syntactically handled like normal transitions but are semantically different as far as they are not taken into account by the firing rule. They are merely a graphical specification for a property that has to be checked *after* the simulation and thus are an extension of the well-known concept of *Facts* as introduced in [GTM76]. In this case the transition **GetLost** (inscribed by a 'K' for *causal chain* - german: Kausalkette) represents the property, that there exists an execution where the causal distance of an item that was consumed from place **Document** by firing transition **GetLost** for a suitable assignment and the produced item on place **Wastebasket** is exactly 5 (i.e. the value inscribed in the transition). To avoid confusion, the arcs connected to query transitions are dashed.

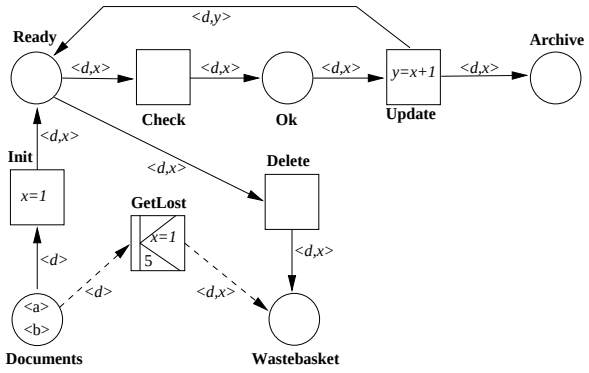

Fig. 7: An example query for our example net

It is impossible to validate this property by looking at occurrence sequences, because there could have occurred arbitrarily many independent events between the precondition and the postcondition of the causal chain - including the creation of new documents. In a simulation which is based on causal semantics we can easily check this property because the causal dependency (and thus the required measure of a *causal distance*) is represented explicitly. Fig. 8 shows what kind of 'pattern' we have to search for in our set of processes.

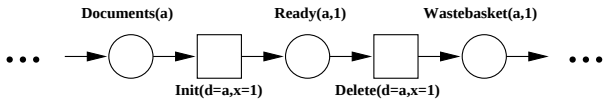

Fig. 8: A pattern that would match the query of Fig. 7

Note that the class of properties requiring a notion of causal distance is only one example where the sequential simulation approach fails. A more general classification will be in the scope of future investigations of the VIP project.

References

[DFOZ96] J. Desel, T. Freytag, A. Oberweis and T. Zimmer. WWW page *VIP - a project overview.* http://www.aifb.uni-karlsruhe.de/pub/InfoSys/VIP.

[DFO97] J. Desel, T. Freytag and A. Oberweis. *Prozesse, Simulation, Eigenschaften netzmodellierter Systeme.* in: Proceedings of Entwurf komplexer Automatisierungssysteme (EKA '97). University of Braunschweig, 1997.

[DO95] J. Desel and A. Oberweis. *Verifikation von Informationssystemen durch Auswertung halbgeordneter Petrinetz-Abläufe.* Technical Report 324, AIFB, Uni Karlsruhe, 1995.

[DOZ96] J. Desel, A. Oberweis and T. Zimmer. *Simulation-based analysis of distributed information system behaviour.* 8th European Simulation Symposium ESS96, Genova, 1996.

[Esp94] J. Esparza. *Model checking using net unfoldings.* Science of Computer Programming, (23):151–195, 1994.

[GTM76] H. Genrich and G. Thieler-Mevissen. *The Calculus of Facts.* Mathematical Foundations of Computer Science, 588–595. Springer-Verlag, 1976.

[JaK89] R. Janicki and M. Koutny. *Towards a theory of simulation for verification of concurrent systems.* in: Parallel Architecture and Languages Europe, LNCS 366, 73–88. Springer-Verlag, 1989.

[Jen92] K. Jensen. *Coloured Petri Nets.* Springer-Verlag, 1992.

[Rei86] W. Reisig. *Petri nets - an Introduction.* Springer-Verlag, 2nd Ed., 1986.