Model Checking Object Petri Nets in Prolog

Berndt Farwer and Michael Leuschel
farwer@informatik.uni-hamburg.de, mal@ecs.soton.ac.uk

Declarative Systems and Software Engineering Group
Technical Report DSSE-TR-2003-4
November 2003

http://www.dsse.ecs.soton.ac.uk/techreports

School of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ, United Kingdom

Model Checking Object Petri Nets in Prolog

(Application Paper)

Berndt Farwer[1] and Michael Leuschel[2]

[1] University of Hamburg, Germany, `farwer@informatik.uni-hamburg.de`
[2] University of Southampton, United Kingdom, `mal@ecs.soton.ac.uk`

Abstract Object Petri nets (OPNs) provide a natural and modular method for the modelling of many real-world systems. We give a structure-preserving translation of OPNs to Prolog, avoiding the need for an unfolding to a flat Petri net. The translation provides support for reference and value semantics, and even allows different objects to be treated as copyable or non-copyable, respectively. The method is developed for OPNs with arbitrary nesting. We then apply logic programming tools to animate, compile and model check OPNs. In particular, we use the partial evaluation system LOGEN to produce an OPN compiler, and we use the model checker XTL to verify CTL formulas. We also use LOGEN to produce special purpose model checkers. We present two case studies, along with experimental results. A comparison to OPN translations to MAUDE specifications and model checking is given, showing that our approach is roughly twice as fast for larger systems. We also tackle infinite state model checking using the ECCE system.

Keywords: Object Petri nets, Model checking, Prolog, Animation, Compilation

1 Introduction

Petri nets are a well-established formalism for modelling and verifying concurrent and reactive systems. Coloured Petri nets (CPNs) were introduced as an extension to Place/Transition (P/T) nets by Jensen [8]. One of the more recent additions to the family of Petri net formalisms are the so-called *object Petri net* formalisms. They allow various structured objects as tokens, including P/T nets or CPNs. When using nets as tokens of such a net, the token nets are often called *object nets* and the CPN whose places contain token nets on the highest level is called the *system net*.

When defining the dynamic behaviour of object Petri nets, a distinction is needed between two fundamentally different kinds of transitions. *Autonomous transitions* can occur in the system net or in an object net. They locally change the marking of the respective net only, i.e., the marking of all other nets are preserved by an autonomous transition firing. On the other hand, there are *synchronisation requirements*, that prevent some transitions from occurring

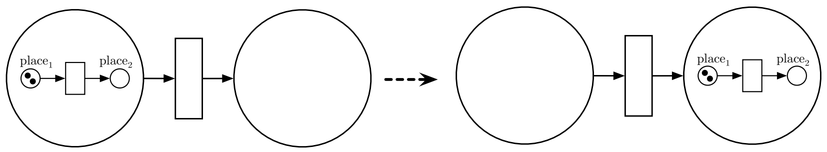


Figure1. Autonomous transition firing in an object Petri net

autonomously. If some transitions are constrained in such a way, they are required to fire simultaneously, subject to the usual (individual) enablement conditions. A simple example of an autonomous system net transition firing is shown in Figure 1. Here the object net is simply moved to a different place of the system net.

Another important attribute of object Petri nets is the existence of various semantics that differ mainly in the treatment of the token nets (e.g. in [5], [23]). *Reference semantics* treat a token net (name) in a place of the system net as a reference to a net instance. Hence, if the same token net (name) appears in different places of the net, they point to the same object net instance. In particular, the marking of each object net instance is applicable for all tokens referring to it. When using *value semantics*, each token net is treated as an individual copy with its own marking.

The advantages of object-based[1] modelling are obvious. Natural objects can be modelled as separate Petri nets, which can be studied to a certain extant even without knowing the context of the system net or other objects in the model. Re-usability is another benefit of such modelling approaches.

Apart from modelling, the issues of validation and verification are central in the study of Petri nets and other formal modelling methods. Most extensions of the basic P/T net formalism use translations to P/T nets, in order to perform model checking or other analysis. While this works fairly well if the model proves to be correct, in the case of the detection of errors, it may not be so straightforward to transfer the result back to the object-based model. Therefore, it would be preferable to do the model checking on an object-based description of the object Petri net. To the best knowledge of the authors, such approach has not yet been pursued for object Petri nets.

The presents transformations of object Petri nets that allow the execution[2] of the net and the application of model-checking techniques. In Section 2 we treat a transformation to a Prolog-style notation with CTL model checking. Section 6 summarises a translation of OPNs to the MAUDE implementation of conditional rewriting logic with LTL model checking.

[1] not to be confused with object-oriented
[2] by execution we mean the automated simulation of a Petri net

2 Translating OPNs to Prolog

In OPNs, for both the system net and the object nets, there are two kinds of transitions: autonomous and synchronised transitions. Different definitions of object Petri nets have been studied in [11], [22], [19], [5], and [6]. We follow a generic approach, that is easily transferable to any of these formalisms.

Let $\mathcal{V}$ be an appropriate set of variables and let $\wr X \int$ denote the set of multisets over the set X. Furthermore, let the arc-weight function $W : (P \times T) \cup (T \times P) \longrightarrow \wr \mathcal{V} \int$ be, such that $W(a, b) = [\![x_1^{a,b}, \ldots, x_{k_{a,b}}^{a,b}]\!]$. Assume a net $\mathcal{N}$ has a set of places $P_{\mathcal{N}} = \{p_1^{\mathcal{N}}, \ldots, p_{n_{\mathcal{N}}}^{\mathcal{N}}\}$.

The treatment of object nets depends to a great extent upon the semantic paradigm used. This shows particularly in the events of fork and join transitions. A fork transition in *value semantics* produces multiple copies of the token nets removed from its input places. Each of the copies can evolve independently. Considering *reference semantics*, on the other hand, would produce multiple references to the same net, so that any evolution of the object net is reflected in all 'copies'. Similar effects have been studied for join transitions.

A translation of an object Petri net has to include information on the synchronisation requirements for its system and object net transitions. Assume $\varrho \in \widetilde{T} \times \widetilde{T}$ is a binary synchronisation relation of the OPN to be encoded. Then include for every $t \in \widetilde{T}$ from the net *net* such that $\neg \exists t' \in \widetilde{T}.(t, t') \in \varrho \vee (t', t) \in \varrho$:

$$\texttt{autonomous}(\mathcal{N}, t).$$

Furthermore, include in the encoding for each transition $t \in \widetilde{T}$ from the net *net* such that $(t, t') \in \varrho \wedge \neg \exists t' \in \widetilde{T}.(t', t) \in \varrho$:

$$\texttt{init\_synch}(\mathcal{N}, t).$$

Note, that the synchronisation relation usually relies on pairwise disjoint sets of names for all object nets and the system net. This requirement does not apply for our encoding, since all transitions are referred to by a net identifier/transition name pair.

2.1 Value Semantics

In this section we describe a transformation of object Petri nets with value semantics into Prolog, and thus suitable for model checking as well as for animation/execution by existing tools (cf. Section 3). Section 2.2 shows the modifications necessary to reflect reference semantics in object Petri nets. The latter is used in the model checking case study of Sections 4–5.

– Assume the system net is named `sn`. For an autonomous *system net transition* t with ${}^{\bullet}t = \{p_1, \ldots, p_{m_t}\}$ and $t^{\bullet} = \{q_1, \ldots, q_{n_t}\}$ include the following DCG code fragment:

```
obj_trans(sn,$t$) -->
    $p_1$ =-=> Token$x^{p_1,t}_{1,1}$,            ...     $p_1$ =-=> Token$x^{p_1,t}_{k_{p_1,t},1}$,
                 ⋮                                          ⋮
    $p_{m_t}$ =-=> Token$x^{p_{m_t},t}_{1,m_t}$,      ...     $p_1$ =-=> Token$x^{p_{m_t},t}_{k_{p_{m_t},t},m_t}$,
    $q_1$ <=+= Token$x^{t,q_1}_{1,1}$,            ...     $q_1$ <=+= Token$x^{t,q_1}_{k_{t,q_1},1}$,
                 ⋮                                          ⋮
    $q_{n_t}$ <=+= Token$x^{t,q_{n_t}}_{1,n_t}$,      ...     $q_{n_t}$ <=+= Token$x^{t,q_{n_t}}_{k_{t,q_{n_t}},n_{n_t}}$ .
```

– For an autonomous *object net transition* t of an object net on with $^\bullet t = \{p_1, \ldots, p_{m_t}\}$, $t^\bullet = \{q_1, \ldots, q_{n_t}\}$ include the same DCG code fragment as for an autonomous system net transition, exchanging only the name sn with on.
– For a transition t of the system net sn which is synchronised with transition t' of an object net with $^\bullet t = \{p_1\}$, $t^\bullet = \{q_1\}$, include:

```
obj_trans(sn,$t$) -->
    p1 =-=> TokenNet,
    {token_trans($t'$,TokenNet,TokenNetAfterFire},
    p2 <=+= TokenNetAfterFire.
```

The encoding is done analogously for other pre- and postsets.
– The *initial marking* is represented as:

```
start([ obj(sn,[        bind($p^{\mathtt{sn}}_1$,[$marking^{\mathtt{sn}}_1$]),
                              ⋮
                        bind($p^{\mathtt{sn}}_{n_{\mathtt{sn}}}$,[$marking^{\mathtt{sn}}_{n_{\mathtt{sn}}}$])]) ]).
```

Here, $marking^{\mathtt{sn}}_i$ denotes a marking of the respective place of the system net. This is encoded as a list containing a finite number of (coloured) tokens and object tokens. For example [b, b, c, obj(net2, [bind(q1, [one]), bind(q2,[])])] represents a marking containing two tokens b, one token c, and an object net net2 with a coloured token of type one in its place q1 and the empty marking in q2.

Note that some more elaborate unification tasks may be required for the case of join transitions involving object nets. These are subject to the precise OPN formalism and cannot not discussed here due to space limitations. Furthermore, our main results rely upon the encoding of reference nets described in the following section, where no further measures have to be taken for join transitions.

2.2 Reference Semantics

In reference semantics, object nets do not occur in any markings. Instead only references to the object nets are used as tokens apart from traditional (coloured) tokens. Some changes have to be applied to the previous encoding, namely:

– The initial marking can contain a finite number of (coloured) tokens and references to object tokens. A reference to an object net `object_net` is represented as `ref(object_net)`. The object net's marking is then given separately in the global marking, for instance by

  ```
  start([ obj(sn,[bind(p1,[ref(low),ref(low), b,b,c]),
                          bind(p2,[b]), bind(p3,[c])]),
          obj(low,[bind(q1,[1,1]),bind(q2,[])])  ]).
  ```

– A synchronisation requirement for a system net transition t with ${}^\bullet t = \{p_1\}$ and $t^\bullet = \{q_1\}$ to be executed in parallel with an object net transition t' can be expressed by the following code fragment, where `TokenNet` and `TokenNetAfterFire` are references to the same object net.

  ```
  obj_trans(sn,t,Ref) -->
      p1 =-=> TokenNet,
      {token_trans(t',TokenNet,TokenNetAfterFire,Ref)},
      p2 <=+= TokenNetAfterFire.
  ```

The encoding of an object Petri net with reference semantics $\mathcal{R}$ will be referred to by $\eta(\mathcal{R})$ in the following.

Lemma 1. *If a transition t of net $\mathcal{R}$ is enabled then its encoding is executable, i.e., there exists a binding satisfying the predicate* `obj_trans(`$\mathcal{N}$`,`t`)` *in its encoding $\eta(\mathcal{R})$.*

Proof. We have to inspect two cases:

(i) enablement of an autonomous transition
(ii) enablement of a set of synchronised transitions

Case (i) again consists of two sub-cases, which are studied below.

System autonomous enablement is limited to those transitions of the system net that do not have a counterpart in the synchronisation relation ϱ. All transitions that have no synchronisation requirement can occur subject to the usual Petri net firing rule. This coincides with the requirement that there are sufficiently many tokens available on the input places of the transition. Since we are not considering place capacities, there are no further requirements in the case of P/T nets.
Autonomous reference net transitions are such that their names do not appear in any synchronisation pair, i.e., they do not have any uplink or downlink inscription. In $\eta(\mathcal{R})$ these transitions are precisely the ones for which a proof of the predicate `autonomous/2` exists, thus enabling the execution of the respective transition's encoding in the presence of an enabling marking. This is provided in the object interpreter by

```
trans(Trans,O,N) :-
    autonomous(NetID,Trans),global_trans(Trans,NetID,O,N).
```

Object autonomous enablement are encoded in the same way as system autonomous transitions. Hence the reasoning from above also holds for this case.

For **case (ii)** let us consider the requirements for the enablement of a transition that appears in the synchronisation relation.

The transition can have one of the following properties:

1. it is invoked by some other transition (uplink)
2. it invokes another transition (downlink)
3. it is both downlink and uplink to some other transitions

In either case, we have ruled out infinite chains and cycles in the synchronisation relation.

The object interpreter governs the execution of synchronised transitions by the predicate `trans/3` which can only be satisfied in the clause

```
trans(Trans,O,N) :-
    init_synch(Name,Trans),global_trans(Trans,net(ID,Name),O,N).
```

The transformation adds the fact
`init_synch(`$\mathcal{N}, t$`).`
for every least element in a transition synchronisation chain. The existence of a least element is guaranteed by $\forall x, y \in \widetilde{T}.(x,y) \in \varrho \rightarrow (y,x) \notin \varrho^+$. For each downlink, an entry is generated in the list representing the obligations that arise from the prospective execution of the transitions encoding. A satisfying binding exists if all transitions involved in the synchronisation are simultaneously enabled.

Thus, a transition's encoding can be executed if the transition is enabled.

Lemma 2. *Any non-synchronised state change in the encoding $\eta(\mathcal{R})$ with respect to the encoded marking of $\mathcal{R}$ corresponds to an autonomous transitions occurrence of the reference net $\mathcal{R}$.*

Proof. The crucial predicate for making changes to the encoded marking in the OPN encoding is `trans/3`.

The definition of this predicate in the object interpreter is:

```
trans(Trans,O,N) :-
    autonomous(Name,Trans),global_trans(Trans,net(ID,Name),O,N).
trans(Trans,O,N) :-
    init_synch(Name,Trans),global_trans(Trans,net(ID,Name),O,N).
```

Hence, any state change with respect to the encoded marking of $\mathcal{R}$ must occur by executing an encoded transition, which can be accomplished either by firing an autonomous transition or by initiating a synchronisation chain.

Lemma 3. *Synchronisation requirements in a reference net $\mathcal{R}$ are correctly reflected in its encoding $\eta(\mathcal{R})$, i.e., a transitions code will only be executed if its synchronisation requirements are met.*

Proof. First, let us note that the code for a transition invoked in a synchronisation cannot be invoked individually. This is due to the requirement that the transition has to satisfy either `autonomous/2` or `init_synch/2` in order for its code to be executed.

The encoding will not provide provability of `autonomous/2` for any transition involved in a synchronisation. Furthermore, for no transition invoked by another transition, will `init_synch/2` be provable. This restricts those transitions' code to be executed in all circumstances apart from an invocation in a synchronisation step.

For the synchronisation requirements we have ruled out the possibility of circular invocations, i.e., a chain of synchronisation requirements may never form a loop. This is formally expressed by the assumption: $\forall x, y \in \widetilde{T}.(x,y) \in \varrho \rightarrow (y,x) \notin \varrho^{+}$. Thus, we avoid infinitary synchronisation requirements.

What remains to be shown is that an enabled synchronisation step in the reference net will indeed lead to its code being executable. This follows directly from Lemma 1 and the fact that the uplinks and downlinks are correctly encoded.

The object interpreter will allow synchronisations to be initialised by the least element in a chain though the following code:

```
trans(Trans,O,N) :-
    init_synch(NetID,Trans),global_trans(Trans,NetID,O,N).
```

Theorem 1. *The encoding $\eta(\mathcal{R})$ of an object Petri $\mathcal{R}$ nets with reference semantics is faithful.*

Proof. Lemmas 2 and 3 state that the behaviour with respect to the encoded transitions and markings in $\eta(\mathcal{R})$ is a subset of the behaviour of $\mathcal{R}$. Furthermore, Lemma 1 states the converse, i.e., any enabled transition can be executed in its encoding.

Thus, we neither loose nor gain any behaviour and the encoding is faithful.

2.3 Combining Value and Reference Semantics

In the XTL model it is easy to combine the concepts of value and reference semantics, to be used in the same model. A simple example of an initial marking mixing the two concepts is given below.

```
start([obj(high_level_net,
       [bind(p1,[ref(low),ref(low),obj(low,[bind(q1,[1]),bind(q2,[])])]),
        bind(p2,[b]), bind(p3,[c])]),
       obj(low,[bind(q1,[1,1]),bind(q2,[])])
     ]).
```

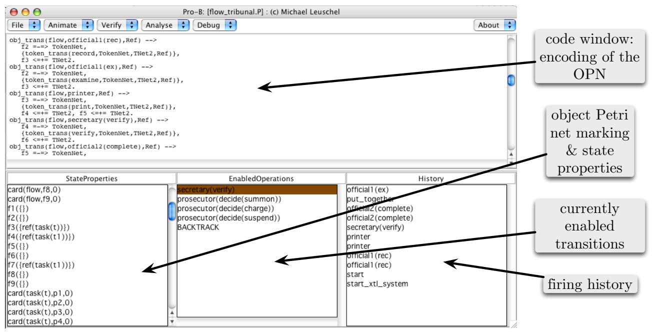
The combination of value and reference semantics makes sense for systems in which there are objects that can be (physically) copied, like an immigration form, and also objects that cannot be copied, like a vehicle in a production line. Both kinds of objects can be concurrently worked on, but only the former can be independently manipulated in a way that the merged information is no longer consistent.

An object net formalism with such combined semantics is the subject of current and ongoing research. From our translation into Prolog and the animation in XTL arises a strong candidate for the defining semantics of such object Petri net extensions.

3 Animating and Compiling

3.1 Animation

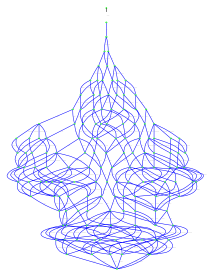
We have applied our generic animator package written in Tcl/Tk and SICStus Prolog, which has been previously used for ProB [13] and a CSP animator [12].


Pro-B: [flow_tribunal.P] : (c) Michael Leuschel
File
Animate
Verify
Analyse
Debug
About
code window: encoding of the OPN
object Petri net marking & state properties
currently enabled transitions
firing history

```
obj_trans(flow,official1(rec),Ref) -->
   f2 =-=> TokenNet,
   {token_trans(record,TokenNet,TNet2,Ref)},
   f3 <=+= TNet2.
obj_trans(flow,official1(ex),Ref) -->
   f2 =-=> TokenNet,
   {token_trans(examine,TokenNet,TNet2,Ref)},
   f3 <=+= TNet2.
obj_trans(flow,printer,Ref) -->
   f3 =-=> TokenNet,
   {token_trans(print,TokenNet,TNet2,Ref)},
   f4 <=+= TNet2, f5 <=+= TNet2.
obj_trans(flow,secretary(verify),Ref) -->
   f4 =-=> TokenNet,
   {token_trans(verify,TokenNet,TNet2,Ref)},
   f6 <=+= TNet2.
obj_trans(flow,official2(complete),Ref) -->
   f5 =-=> TokenNet,
```

StateProperties	EnabledOperations	History
card(flow,f8,0)	secretary(verify)	official1(ex)
card(flow,f9,0)	prosecutor(decide(summon))	put_together
f1({})	prosecutor(decide(charge))	official2(complete)
f2({})	prosecutor(decide(suspend))	official2(complete)
f3({ref(task(t))})	BACKTRACK	secretary(verify)
f4({ref(task(t1))})		printer
f5({})		printer
f6({})		official1(rec)
f7({ref(task(t1))})		official1(rec)
f8({})		start
f9({})		start_xtl_system
card(task(t),p1,0)		
card(task(t),p2,0)		
card(task(t),p3,0)		
card(task(t),p4,0)		

(a) Screenshot of the animator



(b) State space

Figure2. The OPN Animator in action

The interface was initially inspired by [7] and supports (backtrackable) step-by-step animation of the specifications, coverage analysis, as well as visualisation of the state space using the "dot" tool. A screen-shot of the animator can be found in Figure 2(a), while Figure 2(b) shows the state space that it displays for one of our case studies (cf. Section 5.1) consisting of a system net and two object nets. An important aspect of the animator is its ability to cope with non-determinism: all possible choices are presented to the user, if he so wishes, and he can decide upon the exact behaviour. There is also a random mode of animation, where the choices are made by the animator. Finally, the object Petri net execution can be linked with a Java implementation, i.e., the object Petri net "drives" the Java code. One can thus use the object Petri net as a test-case generator for an implementation, or one can use the Java to provide a custom user interface for OPNs, effectively using OPNs for rapid prototyping.

3.2 Compiling by Partial Evaluation

In a sense our translation already compiles OPNs into Prolog. However, using *partial evaluation* we can further improve the efficiency of that translation.

Partial evaluation [9] is a source-to-source program transformation technique which specialises programs by fixing part of the input of some source program P and then pre-computing those parts of P that only depend on the known part of the input. The so-obtained transformed programs are less general than the original but can be much more efficient. The part of the input that is fixed is referred to as the *static* input, while the remainder of the input is called the *dynamic* input.

Partial evaluation has been especially useful when applied to interpreters. In that setting the static input is typically the object program being interpreted, while the actual call to the object program is dynamic. Partial evaluation can then produce a more efficient, specialised version of the interpreter, which is akin to a compiled version of the object program.

Thanks to our translation of object Petri nets into Prolog code, we are able to apply partial evaluation techniques for logic programs to compile object Petri nets into more efficient Prolog code. In particular, we will use the LOGEN system [14], which uses the so-called compiler generator (cogen) approach to specialisation. Figure 3 highlights the way the LOGEN system works. Typically, a user would proceed as follows:

– First the source program is annotated using a binding-time analysis (BTA). This annotated source program can be further edited, by using the LOGEN Emacs mode. This allows a user to manually refine the annotations to make the specialisation more or less aggressive.
– Second, LOGEN is run on the annotated source program and produces a specialised specialiser, called a *generating extension* or also *compiler*.
– This compiler can now be used to specialise the source program for some static input. Note that the same compiler can be run many times for different static inputs (i.e., there is no need to re-run LOGEN on the annotated source program unless the annotated source program itself changes).
  In our case the source program is the object Petri net interpreter and the static input is the encoding of a given object Petri net. The specialised program is then a compiled version (in Prolog) of the object Petri net.
– When the remainder of the input is known, the specialised program can now be run and will produce the same output as the original source program. Again, the same specialised program can be run for different dynamic inputs; one only has to re-generate the specialised program if the static input changes (or the original program itself changes).

After annotation of our interpreter, generation of the object Petri net compiler was very quick (about 240 ms) and compilation itself was also relatively quick (less than half a second for most of our examples, cf. Section 5.3), and could be made faster by making the compiler less aggressive. The improvements for animation speed were about 40 % (speed improvements for verification are much more dramatic due to a decrease in memory usage and specialisation of the model checking component; cf. Section 5.3). The following shows a piece of compiled Prolog code produced by the object Petri net compiler. Basically, every place has become an argument of the specialised procedures and accessing place markings is is thus much faster. The memory usage of the specialised code is also much reduced, as all the encodings of the net structure have been compiled away.

```
compute_trace__4(B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,
       A1,B1,C1,D1,E1,F1,G1,H1,[tribunal(charge)|I1]) :-
    delete__2(ref(task(t1)),I,J1),
    insert__3(ref(task(t1)),J,K1),
    delete__2(L1,F1,M1),
    insert__3(L1,H1,N1),
```

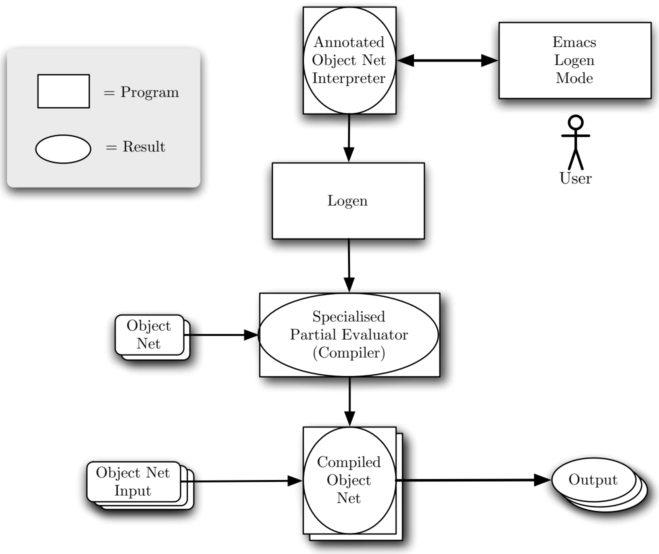


Figure3. Illustrating the LOGEN system and how to compile object Petri nets

```
compute_trace__4(B,C,D,E,F,G,H,J1,K1,K,L,M,N,O,P,Q,R,S,T,U,V,W,
                 X,Y,Z,A1,B1,C1,D1,E1,M1,G1,N1,I1).
```

Thanks to our translation into Prolog, we can also apply other logic programming tools such as termination analysers, type inference tools, verification tools, or static analysers. In the next section, we will show how one can achieve finite and infinite model checking using some of these.

4 Model Checking using Logic Programming Tools

4.1 Finite State Model Checking using XTL

The temporal logic CTL (Computation Tree Logic) introduced by Clarke and Emerson in [2], allows to specify properties of specifications generally described as Kripke structures. The syntax and semantics for CTL are given below.

Given *Prop*, the set of *propositions*, the set of CTL formulae ϕ is inductively defined by the following grammar (where $p \in \mathit{Prop}$):

$$\phi := \mathit{true} \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \forall \bigcirc \phi \mid \exists \bigcirc \phi \mid \forall \phi \mathcal{U} \phi \mid \exists \phi \mathcal{U} \phi$$

A *tabled* Prolog system such as XSB [21] provides very efficient data structures and algorithms to tabulate calls, i.e., it remembers which calls it has already encountered. As was realised in [20] this enables one to write efficient model checkers, with relatively little effort. This has lead to the development of the XMC model checking system, whose performance is comparable to that of SPIN.

Furthermore, as shown in [16], a complete CTL model checker can be written as a relatively simple tabled logic program, called XTL in [17]. Contrary to [20] the aim in [16,17] was not maximum efficiency, but writing a provably correct interpreter that can be fed into existing analysis and optimisation tools.[3] One of the motivations is to use these analysis tools to perform infinite state model checking. Also, XTL is independent of any underlying formalism. It only supposes that the successors of a state s can be computed (through a predicate `trans`) and that the elementary proposition of any state s can be determined (through a predicate `prop`). The interpreter can thus be easily applied to many formalisms, by providing appropriate encodings of `trans` and `prop`. This is exactly the feature we will use to apply to to object Petri nets.

Despite its simplicity and flexibility, XTL has been shown to be on par with some of the most well-known model checkers [17]. Our experiments later in the paper will further underpin this.

4.2 Infinite State Model Checking using ECCE

One of the key issues of model checking of infinite systems is *abstraction*, whereby one approximates an infinite system by a finite one. If proper care is taken, the results obtained for the finite abstraction will be valid for the infinite system.

In earlier work we have tried to solve the abstraction problem by applying existing techniques for the *automatic* control of online partial evaluation. Indeed, in partial evaluation one faces a very similar (and extensively studied) problem: To be able to produce efficient specialised programs, *infinite* computation trees have to be abstracted in a *finite* but also as *precise* as possible way. [15] showed that when we encode Petri nets as logic programs, the specialised programs can be viewed as a finite abstraction of state space covering all (possibly infinite) reachable markings of the Petri net. This allowed one to decide coverability problems (which encompass quasi-liveness, boundedness, determinism, regularity,...) for any Petri net using the specialiser ECCE [18]. Quite surprisingly, the control algorithms behaved very similar to well known Petri net algorithms by Karp–Miller [10] and by Finkel [3]. The advantage of the logic programming approach is that it can in principle be applied to more complicated systems, with richer state structures, provided the system is encoded as a logic program. We can thus apply ECCE to our OPN interpreter and attempt infinite state model checking. Note that ECCE's control algorithms are no longer guaranteed to provide a decision procedure (we may get a "don't know" answer), but as we show later some interesting systems and properties can still be tackled.

[3] These tools work best on declarative programs, and hence the full XMC system is probably not as well suited to analysis and optimisation.

5 Two Case Studies

In Section 5.1 we give a real-world example of a workflow specification. the specification is given as a reference net. Section 5.2 presents some model checking results carried out in XTL on an encoding of that net. Possibilities of optimising these results are provided in Section 5.3. We then present an infinite state case study in Section 5.4 and perform some infinite state model checking in Section 5.5.

5.1 The Prosecution Example

Our example deals with tasks common to many legal systems. The example is taken from [22], where its origin is attributed to W.M.P. van der Aalst. There are several actions involved in this scenario provided by the example. They range from people carrying out sub-tasks (like a secretary verifying some data, an officer filing the case, and a prosecutor deciding upon the legal action to be taken) to more abstract actions (such as a printer producing two copies of a form).

The scenario of a law enforcement agency can be summarised in 7 stages:

1. An offence is filed by officer 1
2. a report and legal form is printed and distributed to be processed by a secretary and another officer, respectively
3. the secretary verifies the details of the report
4. officer 2 fills out the form accompanying the initial report
5. the completed form and report are sent back to officer 1
6. officer 1 now checks whether the completed report and form accurately describe the offence reported in stage 1
7. the prosecutor decides whether the offender be summoned, charged right away, or the case is suspended
8. Appropriate action against the offender is taken or the case is dropped.

It is easy to see that there are dependencies as well as concurrency in this example. For instance stages 3 and 4 are independent of each other, while action 5 relies on both 3 and 4 to have been accomplished prior to 5. Also, there are two actions that are carried out by officer 1. For the model it is desirable not to distribute the possible actions of the same person over the net. Thus, the tasks to be performed by one person are put into the same place and the marking of the task net 'decides' which action, i.e., which transition is chosen.

The overall task is modelled by an object net that represents a protocol of actions to be performed. Each transition in the task net has an uplink that allows it only to occur if a transition with appropriate downlink in the system net is fired simultaneously. The system net and the token net is depicted in Figure 4. Before entering the scenario described above, the system net creates a finite number of tasks (references to token net instances). In Figure 4, the initial marking (a black token in place $p1$) is consumed, two distinct instances of the

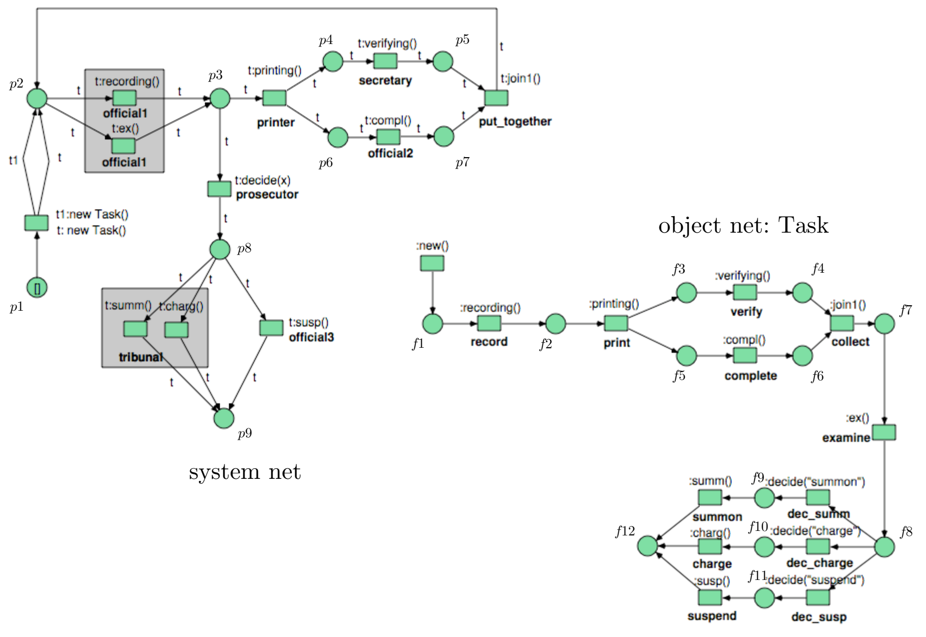


Figure4. System net and object net from the prosecution example

task net are created and uniquely named instances to these nets are produced on the input place to officer 1's transitions.

Note, that the transition printer does not produce two instances of the task on its output places, but rather duplicates the reference to the task net instance, such that each of the tokens produced points to the same instance of the task. Thereby, inconsistencies in the completion of the report and the form are avoided.

5.2 Model Checking Results

Some interesting properties to study in this example are:

- termination of all separate tasks
- termination in the system net with all tasks accomplished
- eventual summoning, suspension, or charging after the case/task has been recorded
- mutual exclusion of the task net's places $f9$, $f10$, and $f11$
- deadlock-freeness.

We analysed these properties using XTL for various numbers of tasks. For example, to find out whether a two task system can reach a state where all tasks are terminated, we check the CTL formula $EF flow.f9 \geq 2$. XTL then produced the following witness trace after 0.02 s: `start`, `official1(rec)`, `official1(rec)`, `printer`, `printer`, `secretary(verify)`, `secretary(verify)`, `official2(complete)`, `official2(complete)`, `put_together`, `official1(ex)`, `put_together`, `official1(ex)`, `prosecutor(decide(summon))`, `prosecutor(decide(summon))`, `tribunal(summon)`, `tribunal(summon)`. A summary of our experiments can be found in Table 1. The timings were taken on a PowerBook G4 1Ghz, 1Gb Ram, running Mac OS X 10.2.6 and XSB Prolog version 2.6 compiled for batched scheduling and early completion. The full state space for the example with 2 tasks which contains 145 states and 337 transitions can be found in Figure 2(b). With 4 tasks it contains 20737 states and 96769 transitions.

Table1. Using XTL to model check the prosecution example

With 2 tasks		
Formula	Result	Time
Overall Termination		
$AFflow.f9 = 2$	true	0.289 s
$EFflow.f9 \geq 2$	true	0.020 s
$EFflow.f9 = 2$	true	0.019 s
$EFflow.f9 \geq 3$	false	0.18 s
Mutual Exclusion		
$EF(t.p9 \geq 1 \wedge t.p10 \geq 1)$	false	0.219 s
$EF(t.p9 \geq 1 \wedge t.p11 \geq 1)$	false	0.201 s
$EF(t.p10 \geq 1 \wedge t.p11 \geq 1)$	false	0.219 s
$EFt.p9 \geq 2$	false	0.1910 s
$EF(t.p9 \geq 1 \wedge t1.p10 \geq 1)$	true	0.219 s
Termination of all Tasks		
$AFt.p12 = 1$	true	0.270 s
$AFt1.p12 = 1$	true	0.310 s
$AFAGt1.p12 = 1$	true	0.650 s

With 4 tasks		
Formula	Result	Time
$EFflow.f9 \geq 4$	true	0.071 s
$EFflow.f9 = 4$	true	0.05 s
$EFflow.f9 \geq 5$	false	47.779 s

With 5 tasks		
$EFflow.f9 = 5$	true	0.091 s

With 6 tasks		
$EFflow.f9 = 6$	true	0.13 s

5.3 Optimising using partial evaluation

As the experiments have shown, XTL performs quite well on the OPN interpreter, checking more than 400 states per second. However, there are various ways to substantially improve the model checking performance. First, XTL can be used in a mode where the counter example trace is not constructed as a Prolog term during the model checking. For technical reasons[4] this is more efficient, as can be seen in the "No Trace" column of Table 2. Note that the counter example can still be extracted from the XSB table structures [4].

Second, we can apply our compilation techniques of Section 3.2. In fact, we can now not only specialise the OPN interpreter but the model checker as well,

[4] We can use XSB in "local scheduling" rather than in batched mode with early completion.

Table2. Using Logen to speedup model checking

Formula	Result	With Trace runtime	No Trace runtime	After Logen runtime	Logen Speedup	Total Speedup
With 2 tasks						
$AF flow.f9 = 2$	true	0.289 s	0.260 s	0.01 s	26.0	28.9
$EF flow.f9 \geq 3$	false	0.18 s	0.08 s	0.01 s	8.0	18.0
With 4 tasks						
$AF flow.f9 = 4$	true	83.63 s	78.40 s	6.62 s	11.8	12.6
$EF flow.f9 \geq 5$	false	47.779 s	19.11 s	5.691 s	3.36	8.4

i.e., we can specialise the model checker for a particular temporal logic formula and for using our OPN interpreter for a particular object Petri net. This is what we have undertaken, and the results can be found in Table 2. Note that we have derived the compiler from the model checker that does not compute traces (but it would have been possible to do so for the model checker that does compute them). For the second example, the specialised model checker was thus able to explore 3644 states and 17004 transitions per second.

Note that generating the object Petri net/temporal logic compiler took 270 ms (a compiler that can be used for any CTL temporal logic formula and any object Petri net encoding). Compilation for the formula $EF flow.f9 \geq 5$ and the object Petri net with 4 tasks took 460 ms. So, counting the 460 ms, compiling still gives a very respectable 3.11 times improvement in speed. One can reduce compilation time by making the specialisation less aggressive (e.g., if one works mainly on small examples). Also, if one compiles the same object Petri net for different formulas, not all of the compilation has to be redone Finally, it is interesting to note that the more complicated AF formulas lead to bigger speedups, as more of the model checking component is specialised.

5.4 Immigration Example

Figure 5 depicts the components of an OPN representing the usual immigration procedure. It involves

- receiving a passport
- applying for a visa (single entry or lifetime)
- entering the country
- leaving the country
- re-entering the country
- re-applying for a visa

This system is infinite state, so without further restriction it cannot be analysed using finite state model checkers such as XTL (due to their depth-first exploration, they will often not find counterexamples). We will thus now attempt to apply the ECCE tool, as outlined in Section 4.2.

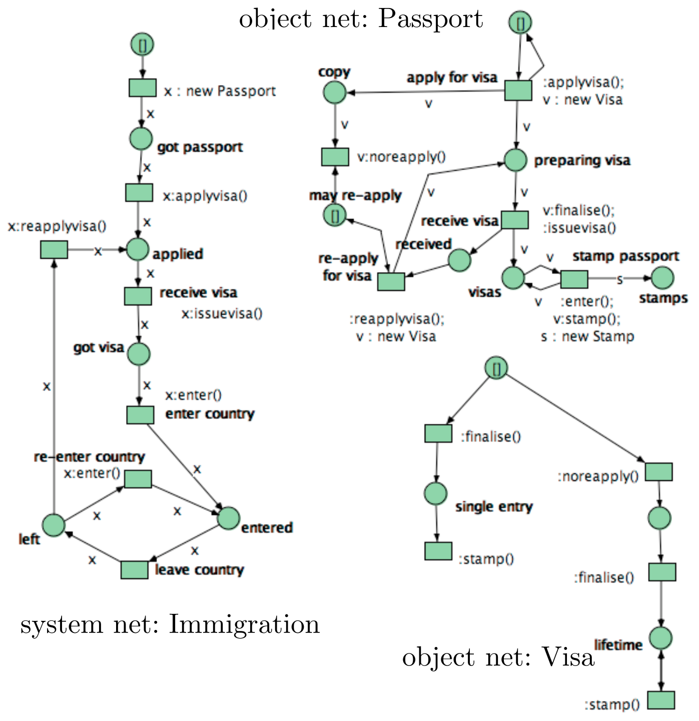


Figure5. Object Petri net for the immigration example

5.5 Infinite Model Checking

We have attempted to prove safety properties of the immigration example. For example, we have attempted to prove that the property $EF immigration.i5 = 2$ is false. For that, we have first applied LOGEN to compile the OPN and the temporal logic formula. We have then applied ECCE on the resulting compiled program, which resulted in the following (where `sat|__1` was the entry point of the compiled program):

```
/* Specialised program generated by Ecce 1.1 */
/* Transformation time: 758820 ms */
/* Specialised Predicates:

sat__1(immi,[1],[0],[],[],[],[],[]) :-  fail.
```

This was achieved using the default settings of ECCE (not tuned for model checking) using the most-specific version post-processing. ECCE was hence able to prove that our safety property cannot be violated! In order to achieve that, ECCE had effectively produced a symbolic abstraction of the infinite state space, consisting of 19 nodes (not counting many intermediate nodes that were thrown away); the most-specific post-processor was then able to conclude that none of these symbolic nodes could violate the safety property.

We have attempted another safety property, and have been equally successful.

6 Comparison with Maude

In recent attempts to find a unifying framework for concurrency formalisms, the theory of rewriting logic has turned out to be very useful. P/T nets, coloured Petri nets, and algebraic Petri nets amongst other formalisms have been successfully modelled in Maude.

Maude is a general tool for formal specification and analysis that is based on a very efficient rewriting engine. Typically, the Maude specifications are executable, though there is no nice GUI attached to the command line-based tool.

We use (conditional) rewriting rules to represent the OPN transitions, specify properties, and prove them with the LTL model checker. Downlinks of the OPN's transitions are modelled directly in the conditional part of their rewrite rules.

The translation of OPNs to equivalent rewrite specification is very straightforward. Due to the lack of space, we do not give a formal translation in this paper, but rather study an example of a transition from the OPN depicted in Figure 4.

Consider, for instance, the printer transition of the system net. It is encoded as:

```
crl [FE-printer]:   fe-pool-2(on NM1) fe-for-secretary(NM2)
                          fe-for-official-2(NM3) Net(on,M1) =>
                    fe-pool-2(NM1) fe-for-secretary(on NM2)
                      fe-for-official-2(on NM3) Net(on,M2)
                    if M1 rprinting => M2 fprinting .
```

The string in square brackets supplies a name for reference in the traces of the rewriting process. We are faced with a conditional rule (`crl`) that rewrites the system state where the `place fe-pool-2` contains (possibly among others) a token `on` such that `on` is the name of an object net. The effect of the rewrite is that the reference `on` is removed from `fe-pool-2` and copies of the reference are generated on `fe-for-secretary` and `fe-for-official-2`. This operation does not change any other tokens in the system net, but the rewrite can only be executed if the conditional is satisfied and thus provides the successor marking `M2` of the object net `on`.

The conditional involves the downlink `printing` of the system net transition. This is represented here as an additional token that is provided only in this conditional. It allows the object net transition print's rule to be executed in the encoding:

```
rl [print]: task-recorded rprinting =>
                 task-printed-1 task-printed-2 fprinting .
```

This rule can only be executed, if the 'token' `rprinting` is provided. It then produces – in addition to the tokens required by the modelled transition – a unique 'token' `fprinting`, which is also handed over to the condition of the invoking system net rule. This mechanism ensures the synchronisation of the two transitions.

The rewrite specification uses multiset sorts for the system net markings and the object net markings. The synchronous channels (downlinks and uplinks) are also treated as markings in the above sense.

As the model checker in MAUDE works on LTL, in Table 3 we only give figures for properties that can be expressed both in LTL and CTL to allow a comparison with the results of Section 5.2. Note that the size of the state spec for 5 tasks was 248,832 nodes, so XTL manages to process more than 2000 nodes per second. The table does not include the time to compile the OPNs. But, even including the compile time of 0.72 s for the 5 task net, XTL + LOGEN is still 1.88 times faster. However, for small systems compilation via LOGEN is not really beneficial: for the 2 task net compiling takes 0.22 s and it is thus better to use the unspecialised model checker which runs in 0.289 s (unless one wants to check many formulas for the 2 task net; in which case compilation may still be beneficial).

Table3. Using MAUDE to model check the prosecution example

No of Tasks	Formula	Result	MAUDE	XTL + LOGEN	Factor
2	$AF flow.f9 = 2$	true	0.04 s	0.01 s	4
4	$AF flow.f9 = 4$	true	15.70 s	6.62	2.37
5	$AF flow.f9 = 5$	true	218.61 s	115.65	1.89

We come to the conclusion that the conversion to Prolog is fairly easy, roughly as straightforward as the conversion to MAUDE, but in some crucial aspects the XTL model checker is much faster than the MAUDE model checker. Especially, the possibility of applying model checking techniques to certain infinite state systems should not be under-estimated.

7 Outlook and Conclusion

In summary, the main contributions of this paper are:

- Animation and execution of object Petri nets via a structure-preserving transformation to Prolog, where the original structure of the OPNs are not lost. This allows easy translation of animation or verification results back to the original OPNs. We have also shown how to model both reference and value semantics-based OPNs.
- We have shown how to apply an offline partial evaluation tool to automatically compile OPNs into efficient Prolog code and have produced an OPN

compiler. This allows fast animation, but also opens up the possibilities to use OPNs for rapid prototyping or maybe even full runtime execution.

- We have shown how to do efficient CTL model checking of object Petri nets by applying the Prolog based XTL on our OPN interpreter,
- We have used partial evaluation to further improve the efficiency of model checking by producing special purpose model checkers,
- We have applied this to a non-trivial example from the literature and have compared our results with MAUDE, showing the efficiency of our approach
- We have used the ECCE tool to perform infinite state model checking on another example.

Among the greatest limitations of many simulation tools is the necessity of applying strategies to the execution engine, in order to achieve fairness. Depending on the tool and formalism this can be very hard to implement, and more importantly the strategy is a further component that needs to be verified or at least validated. Using model-checking techniques, we avoid these problems altogether by inspecting the full state space of the system model. Though restricted by the nature of model checking, the method outlined in this paper is, e.g., applicable to a large class of real-world workflow systems, business processing systems, and manufacturing systems.

Future work The translation of object Petri nets (and reference nets) is completely canonical, so that an export of a net specification in this format from the Renew tool will pose no problem. The plug-in architecture of Renew should allow direct interaction with the model checker in a future release.

Due to the translation into Prolog, we can now integrate OPNs with other formalisms for which similar interpreters have already been written, such as CSP [12] or B [13]. A promising avenue would be linking OPNs with B: indeed B provides very good, high-level data modelling but it lacks modelling of concurrency aspects. This has lead to researchers trying to combine B with CSP [1]. However, OPNs would also be an attractive formalism, where B machines would be naturally passed as tokens. This would also support multiple B machines; something which current CSP/B approaches cannot do. Finally the translation into Prolog also opens up the possibilities to extend OPNs with new features such as constraints, or adapt them for specific application domains.

References

1. M.J. Butler. csp2B: A practical approach to combining CSP and B. *Formal Aspects of Computing*, 12:182–198, 2000.
2. E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer-Verlag.
3. A. Finkel. The minimal coverability graph for Petri nets. *Lecture Notes in Computer Science*, 674:210–243, 1993.

4. Hai Feng Guo, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justification based on program transformation. In Michael Leuschel, editor, *Logic-based Program Synthesis and Transformation (LOPSTR'2002)*, LNCS 2664, pages 158–159, Madrid, Spain, September 2002. Springer-Verlag.
5. B. Farwer. Comparing concepts of object Petri net formalisms. *Fundamenta Informaticae*, 47(3–4):247–258, 2001.
6. B. Farwer and K. Misra. Modelling with hierarchical object Petri nets. *Fundamenta Informaticae*, 55(2):129–147, 2003.
7. P. Henderson. Modelling architectures for dynamic systems. In A. McIver and C. Morgan, editors, *Programming Methodology*. Springer-Verlag, 2003.
8. K. Jensen. An Introduction to High-Level Petri nets. Technical Report ISSN 0105-8517, Department of Computer Science, University of Aarhus, October 1985.
9. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
10. R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3:147–195, 1969.
11. C. A. Lakos. Object Petri nets—definition and relationship to coloured nets. Technical report, TR94-3, Computer Science Department, University of Tasmania, 1994.
12. Michael Leuschel. Design and implementation of the high-level specification language csp(lp) in prolog. In I. V. Ramakrishnan, editor, *Proceedings of PADL'01*, LNCS 1990, pages 14–28. Springer-Verlag, March 2001.
13. M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
14. M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming*, 4(1):139–191, 2004.
15. Michael Leuschel and Helko Lehmann. Solving coverability problems of Petri nets by partial deduction. In Maurizio Gabbrielli and Frank Pfenning, editors, *Proceedings of PPDP'2000*, pages 268–279, Montreal, Canada, 2000. ACM Press.
16. Michael Leuschel and Thierry Massart. Infinite state model checking by abstract interpretation and program specialisation. In Annalisa Bossi, editor, Logic-Based Program Synthesis and Transformation. *Proceedings of LOPSTR'99*, LNCS 1817, pages 63–82, Venice, Italy, 2000.
17. Michael Leuschel and Thierry Massart. Logic programming and partial deduction for the verification of reactive systems: An experimental evaluation. In Gethin Norman, Martha Kwiatkowska, and Dimitar Guelev, editors, *Proceedings of AVoCS 2002, Second Workshop on Automated Verification of Critical Systems*, pages 143–149, 2002. Available as Technical Report CSR-02-6, University of Birmingham.
18. M. Leuschel, B. Martens, and D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.
19. I. A. Lomazova. Nested Petri nets — a formalism for specification of multi-agent distributed systems. In H.-D. Burkhard, L. Czaja, H.-S. Nguyen, and P. Starke, editors, *Concurrency Specification and Programming (CSP'99), Proceedings*, pages 127–140. University of Warsaw, 1999.
20. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Terrance Swift, and David S. Warren. Efficient model checking using tabled resolution. In O. Grumberg, editor, *Proceedings CAV'97*, LNCS 1254, pages 143–154. Springer-Verlag, 1997.

21. K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453, Minneapolis, Minnesota, May 1994. ACM.
22. R. Valk. Petri nets as token objects. An introduction to elementary object nets. In J. Desel and M. Silva, editors, *Applications and Theory of Petri Nets 1998. Proceedings*, volume 1420, pages 1–25. Springer-Verlag, 1998.
23. R. Valk. Relating different semantics for object Petri nets. Technical report, FBI-HH-B-266/00, Fachbereich Informatik, Universität Hamburg, 2000.

A The Object Interpreter

```
/* interpreter for object petri nets, both for
  value and reference semantics */

/* exported predicates for XTL:
    trans/3,
    prop/2   */

trans(Trans,O,N) :-
    autonomous(Name,Trans),global_trans(Trans,net(ID,Name),O,N).
trans(Trans,O,N) :-
    init_synch(Name,Trans),global_trans(Trans,net(ID,Name),O,N).

global_trans(Trans,Net,Env,NEnv) :-
    global_trans2(Trans,Net,Env,E2,Pending),
    perform_pending_actions(Pending,E2,NEnv).

global_trans2(Trans,net(ID,Name),[obj(net(ID,Name),Marking)|T],
                  [obj(net(ID,Name),NMarking)|T],Pending) :-
    obj_trans(Name,Trans,Pending,Marking,NMarking).
global_trans2(Trans,Net,[H|T],[H|TT],Pending) :-
    global_trans2(Trans,Net,T,TT,Pending).

perform_pending_actions([],Env,Env).
perform_pending_actions([H|T],Env,NEnv) :-
   perform_pending_actions(H,Env,E2),
   perform_pending_actions(T,E2,NEnv).
perform_pending_actions(ref(Trans,RefNet),Env,NEnv) :-
      /* print(ref(Trans,RefNet)),nl, */
      global_trans(Trans,RefNet,Env,NEnv).
perform_pending_actions(add(Net),Env,NEnv) :- add_net(Net,Env,NEnv).

add_net(NetID,[obj(ID,M)|T],[obj(ID,M2)|AT]) :-
   ((NetID=ID)
     -> (start_marking(NetID,M2),T=AT) /* net already exists */
     ;  (M2=M,add_net(NetID,T,AT))   ).
add_net(NetID,[],[NewNet]) :- new_net(NetID,NewNet).

prop(L,P) :- member(Net,L), prop2(Net,P).
```

```
prop2(obj(Net,Marking),card(Net,Place,Card)) :-
   member(bind(Place,Val),Marking), len(Val,Card).
prop2(obj(Net,Marking),card_geq(Net,Place,Card)) :- nonvar(Card),
   member(bind(Place,Val),Marking), len(Val,L), L >= Card.
prop2(obj(_,Marking),P) :-
   member(bind(Id,Val),Marking),  P =.. [Id,Val].
prop2(obj(_,Marking),P) :-
   member(bind(Id,Val),Marking), member(obj(Net,MN),Val),
   prop2(obj(Net,MN),NP), P =.. [Id,Net,NP].

len([],0).
len([_|T],L) :- len(T,L1), L is L1+1.

token_trans(Trans,obj(net(ID,Name),Marking),obj(net(ID,Name),M2),
    Ref) :-   obj_trans(Name,Trans,Ref,Marking,M2).
token_trans(Trans,ref(NetID),ref(NetID),ref(Trans,NetID)).

new_net(NetID,obj(NetID,Marking)) :- start_marking(NetID,Marking).

start_marking(net(ID,Name),Marking) :-
   findall(bind(PlaceName,DefaultValue),
           place(net(ID,Name),PlaceName,DefaultValue), Marking).

:- op(500,yfx,'=-=>').   :- op(500,yfx,'=*=>').
:- op(500,yfx,'<=+=').   :- op(500,yfx,'<=*=').

/* delete_all_tokens */
'=*=>'(Place,Tokens,Marking,AfterMarking) :-
  lookup_value(Place,Marking,Tokens),
  Tokens \= [],
  store_value(Place,[],Marking,AfterMarking).

/* delete_one_token */
'=-=>'(Place,Token,Marking,AfterMarking) :-
  lookup_value(Place,Marking,MultiSet),
  delete(Token,MultiSet,NewMS),
  store_value(Place,NewMS,Marking,AfterMarking).

/* add_all_tokens */
'<=*='(Place,Tokens,Marking,AfterMarking) :-
  lookup_value(Place,Marking,MultiSet),
  insert_all(Tokens,MultiSet,NewMS),
  store_value(Place,NewMS,Marking,AfterMarking).

/* add_one_token */
'<=+='(Place,Token,Marking,AfterMarking) :-
  lookup_value(Place,Marking,MultiSet),
  insert(Token,MultiSet,NewMS),
  store_value(Place,NewMS,Marking,AfterMarking).
store_value(Var,Value,[],[bind(Var,Value)]) :-
   print('!WARNING: Identifier not yet defined: '), print(Var),nl.
store_value(Var,Value,[bind(Var,_)|T],[bind(Var,Value)|T]).
store_value(Var,Value,[bind(V,VV)|T],[bind(V,VV)|UT]) :-
  \+(Var=V), store_value(Var,Value,T,UT).

lookup_value(Id,State,Val) :-
 (member(bind(Id,Val),State) -> true
    ; (print('!ERROR: Identifier does not exist for lookup: '),
       print(Id),nl,
       fail)   ).

member(X,[X|_]).
member(X,[_|T]) :- member(X,T).

delete(X,[X|T],T).
delete(X,[H|T],[H|RT]) :- delete(X,T,RT).

insert_all([],MS,MS).
insert_all([H|T],InMark,OutMark) :-
   insert(H,InMark,Int), insert_all(T,Int,OutMark).

insert(X,[],[X]).
insert(X,[H|T],Res) :-
  ( (X @=< H) -> (Res = [X,H|T]) ; (Res = [H|RT], insert(X,T,RT))).
insert(X,Z,[X,Z]) :- Z \= [], Z\=[_|_],
    print('!type error, place not a list: '), print(Z),nl.
```