Eindhoven University of Technology

Department of Mathematics and Computing Science

Z and high level Petri nets

by

K.M. van Hee L.J. Somers M. Voorhoeve

Computing Science Note 91/23
Eindhoven, September 1991

Copies can be ordered from:
Mrs. F. van Neerven
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

Z and high level Petri nets

K.M. van Hee L.J. Somers M. Voorhoeve

Department of Mathematics and Computing Science
Eindhoven University of Technology
Den Dolech 2, P.O. Box 513
5600 MB Eindhoven, the Netherlands
Email: wsinlou@win.tue.nl

Abstract

High level Petri nets have tokens with values, traditionally called colors, and transitions that produce tokens in a functional way, using the consumed tokens as arguments of the function application. Large nets should be designed in a top-down approach and therefore we introduce a hierarchical net model which combines a data flow diagram technique with a high level Petri net model. We use Z to specify this net model, which is in fact the metamodel for specific systems. Specific models we specify partly by diagrams and partly in Z. We give some advantages and disadvantages of using Z in this way. Finally we show how to specify systems by means of an example.

1 Introduction

The last years have shown a growing interest in the formal specification of distributed systems. Such a formal description should take care of the distribution aspects, the interaction between the distributed parts, the transitions between successive states of the system and the state space itself.

Petri nets, see e.g. [Jensen 91], have been used for quite a while to specify concurrent distributed systems. These nets have been augmented recently by a hierarchy to allow a systematic top-down design of a system specification. Such a design method is very similar to the common (informal) use of data flow diagrams [Yourdon 89].

Colored nets make it possible to attach values to tokens. For the specification of these values and the functionality of the transitions one needs a specification language. Usually a functional language is used for the specification of the transition functions. Recently, a few tools have been developed that offer hierarchical colored Petri nets as a specification formalism, cf. [Albrecht 89] and [Hee 89]. Our tool, ExSpect, is based upon a hierarchical timed net model and a functional language. This system has been in use for two years and we have gained a lot of experience in practical applications, e.g. [Aalst 90].

On the other hand, formalisms like Z and VDM are used frequently to specify reactive systems. They do not have mechanisms for treating concurrency and distribution in a straightforward way. However, a formalism like Z seems to be very well suited to specify the transitions in a colored Petri net, thereby replacing the functional language normally used.

There have been more attempts to integrate Z with graphical languages for the description of distributed systems. For instance, for HOOD such an integration is considered. In [Giovanni 90] it is noted that an integration with Petri nets is a point of research.

In the next sections we will use Z in two ways. Firstly for defining what a hierarchical Petri net is (section 2) and secondly for the specification of the state transitions of a Petri net (section 3). We will follow the notation of [Spivey 89]; if not we make a remark.

2 Hierarchical net model

Here we introduce the hierarchical net model that is being used in ExSpect [Hee 89] and that is closely related to hierarchical CP-nets [Jensen 91]. First, we define so called *flat nets*, which are in fact ordinary colored Petri nets. Then we define *hierarchical nets* and show how such a net determines a flat net.

We use the Z notation for the above definitions, thereby showing that Z is not only useful for modeling specific systems, but also for defining metamodels or model types. As observed in [Diepen 90], Z schemas can be interpreted as implicitly defined tables. So, if we give a schema for the net model, every flat net corresponds to one and only one element or tuple of the table determined by the schema.

2.1 Flat nets

A flat net has four basic types

$$[\mathit{Place}, \mathit{Transition}, \mathit{Connection}, \mathit{Value}].$$

The state of a flat net is determined by objects called *tokens*. Tokens reside at a place, an element of the type *Place*, and have a value belonging to the type *Value*. State changes are caused by transitions, elements of the type *Transition*. A transition can *fire*, consuming tokens from a specific set of places and producing tokens for another (not necessarily disjoint) set of places. A transition has a set of input and a set of output connections (elements of the type *Connection*). Connections are used to connect transitions to places.

A *flat net structure* is a directed labeled bipartite graph with places and transitions as nodes and connections as arcs. Traditionally, places are represented by circles and transitions by rectangles (cf. figure 1).

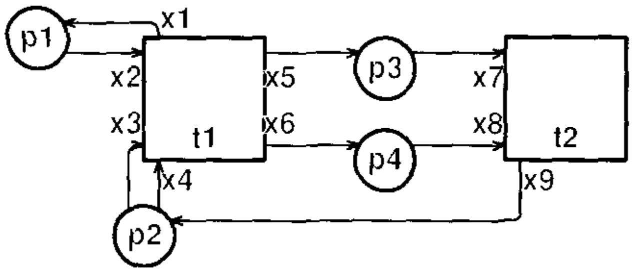

Figure 1: An example of a Petri net.

A flat net structure is defined by the following schema

FlatNetStructure

$P : \mathbb{P}\,Place$
$T : \mathbb{P}\,Transition$
$I : Transition \nrightarrow \mathbb{P}\,Connection$
$O : Transition \nrightarrow \mathbb{P}\,Connection$
$M : Connection \nrightarrow Place$

$\mathrm{dom}(I) = \mathrm{dom}(O) = T \wedge$
$(\bigcup \mathrm{rng}(I)) \cap (\bigcup \mathrm{rng}(O)) = \mathrm{dom}(M) \wedge$
$\mathrm{rng}(M) = P \wedge$
$\forall t_1, t_2 : T \mid t_1 \neq t_2 \bullet I(t_1) \cap I(t_2) = O(t_1) \cap O(t_2) = I(t_1) \cap O(t_2) = I(t_1) \cap O(t_1) = \emptyset$

A net is determined by a set of places P, a set of transitions T, two functions I and O that assign to each transition t a set of connections through which it *consumes* tokens ($I(t)$) and a set of connections through which it *produces* tokens ($O(t)$). Each connection is assigned to a place [Hee 91]. So, when a transition t fires, it consumes as many tokens from a place as there are connections of $I(t)$ assigned to it. Furthermore, it produces for each connection in $O(t)$ one token or none, so if there are n connections in $O(t)$ assigned to a place, t may produce any number of tokens between 0 and n for this place. The last condition in the schema states that all connections are unique. In practice, we require only that connections between the same place and transition with the same direction have different labels. In that case we augment the label with a pair (t, p) for output connections and (p, t) for input connections (where t and p are the transition and the place involved) to obtain a unique label.

It is easy to define an instance of the type FlatNetStructure that describes e.g. the graph of fig. 1.

The flat net structure describes only the *static* aspects of a net model. The *dynamics* of such a model are described by an (unlabeled) *transition system.* Such a transition system has a state space S and a binary relation Tr over S. Each pair of Tr corresponds to a possible state transition of the whole system. In fact the transition system of the net model may be considered as an *operational semantics* of the flat net structure.

We define a derived type called *State*,

$$State == Place \nrightarrow (Value \nrightarrow \mathbb{N}).$$

So a state is a mapping that assigns to places a function counting the number of tokens per value. (Note that this function does not represent a bag since we allow numbers to be zero; however it is equivalent to one.)

For each place in a net structure, we define a subset of *Value* that represents the set of allowed token values for this place. We call this set the value *domain* of the place. In the next schema we define the *state space* of a net structure

$$\begin{array}{|l}
\multicolumn{1}{l}{StateSpace} \\ \hline
P : \mathbb{P} Place \\
S : \mathbb{P} State \\
D : Place \nrightarrow \mathbb{P}\, Value \\ \hline
\mathrm{dom}(D) = P \wedge \\
\forall s : S \bullet \mathrm{dom}(s) = P \wedge \\
\forall s : S \bullet \forall p : P \bullet \mathrm{dom}(s(p)) \subset D(p) \\ \hline
\end{array}$$

So component S defines the state space and D the function that assigns domains to places. The last predicate states that tokens at a certain place should have values in the domain of that place.

The next schema combines the state space and flat net structure schemas, adding two components. The first component is a function L that assigns to each transition a *local state transition function* which formalizes the consumption and production of tokens. The second component is the transition relation Tr over the state space.

$$\begin{array}{|l}
\multicolumn{1}{l}{NetModel} \\ \hline
FlatNetStructure \\
StateSpace \\
L : Transition \nrightarrow (State \nrightarrow State) \\
Tr : \mathbb{P}(State \times State) \\ \hline
1 \;\; \mathrm{dom}(L) = T \\
2 \;\; \forall t : T \bullet \mathrm{dom}(L(t)) \subset S \\
3 \;\; \forall t : T \bullet \forall s : \mathrm{dom}(L(t)) \bullet \forall p : \mathrm{dom}(s) \bullet \\
\qquad \Sigma(s(p)) = \#\{c : I(t) \mid M(c) = p\} \\
4 \;\; \forall t : T \bullet \forall s : \mathrm{dom}(L(t)) \bullet \forall p : \mathrm{dom}(L(t)(s)) \bullet \\
\qquad \Sigma(L(t)(s)(p)) \leq \#\{c : O(t) \mid M(c) = p\} \\
5 \;\; \forall x : Tr \bullet \mathrm{first}(x) \in S \wedge \mathrm{second}(x) \in S \\
6 \;\; \forall s_1, s_2 : S \mid (s_1, s_2) \in Tr \bullet \exists t : T \bullet \exists s : S \bullet \\
\qquad \forall p : P \bullet s(p) \leq s_1(p) \wedge s \in \mathrm{dom}(L(t)) \wedge \\
\qquad \forall p : P \bullet s_2(p) = s_1(p) - s(p) + L(t)(s)(p) \\ \hline
\end{array}$$

The third rule states that for each input connection one token is consumed; the fourth rule that for each output connection at most one token is produced. Rule six tells us how a (global) state transition is derived from a local state change caused by a transition in the net structure. Note that we add, subtract and compare functions; the generic function Σ summarizes the images of a function.

Given an initial state, the net model determines the set of paths of the net according to the transition law. If this law is deterministic, i.e. Tr is in reality of the type $State \nrightarrow State$, there is only one path. Note that for each state transition precisely one net transition should fire.

2.2 Hierarchical nets

Now we can introduce *net hierarchy*. For large systems it is too complicated to define a flat net structure in one step. One often uses a top-down design method similar to data flow diagram techniques [Yourdon 89] or HOOD [Giovanni 90].

Our hierarchical net structure allows us to decompose nets into places and *subnets* or *agents*. (Also compare channel/agency nets [Reisig 87]). At each level of the decomposition process we have to decide which subnets are *elementary* (i.e. are transitions) and which ones need further decomposition. When all subnets have been decomposed into elementary ones, we can proceed by defining a local state transition function for each elementary subnet.

The above hierarchical net structure is useful not only for the design process but also for documenting the net model, since it provides an overview of the flat net structure.

A subnet usually has connections like a transition. We always assume that the (one) subnet at the top of the hierarchy has no connections anymore. This means that if we want to describe a system that communicates with an environment (by exchanging tokens), we have to model that environment as a subnet and incorporate it at the top level. (Note that this was already the case in the flat net structure.) Figure 2 shows three levels of an example hierarchy.

We see that every connection of a subnet is assigned to a place or connection in its supernet. In figure 2, y is assigned to x and x to z. In the next schema we define the hierarchical net structure, generalizing the flat net structure. Each subnet (except the top level one) and each place are mapped to precisely one supernet. These mappings constitute the components HN and HP in the schema. Component Top denotes the top level net. For subnets as well as transitions we use the same type

$$[Net]$$

replacing the earlier type *Transition*. Component N denotes all subnets, including transitions. Components MP and MC denote the assignment of connections. They replace component M of the flat net structure.

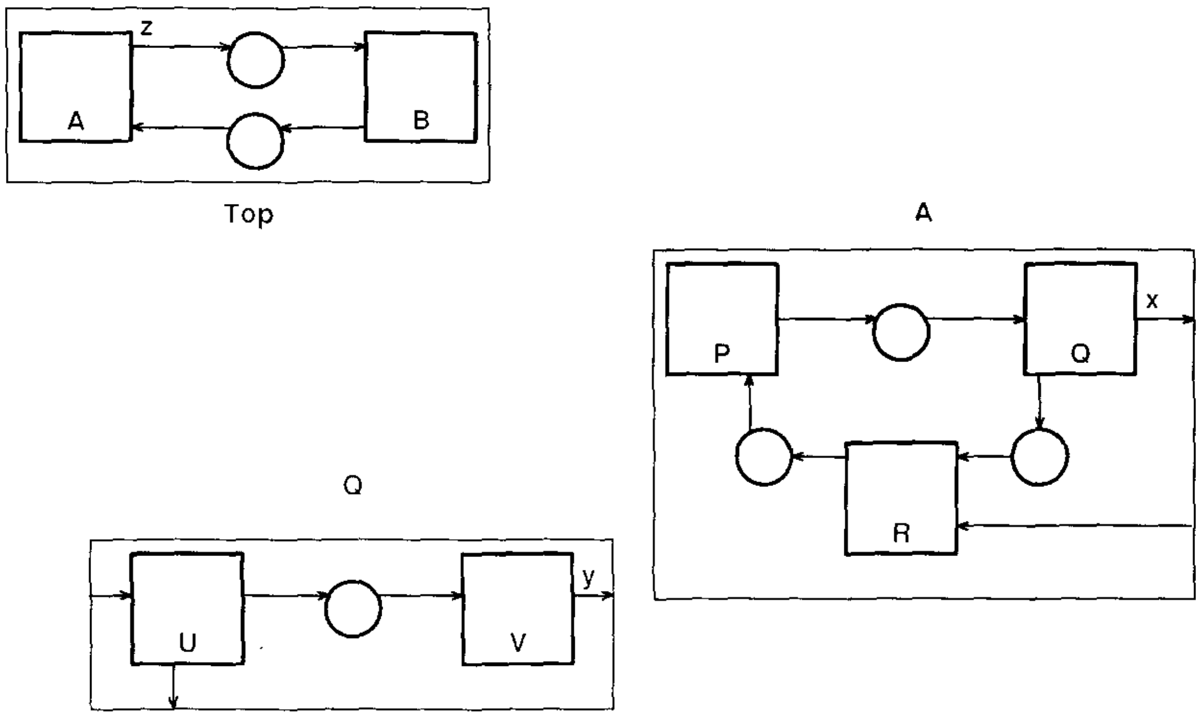

Figure 2: Hierarchical decomposition.

HierarchicalNetStructure

$P : \mathbb{P}\,Place$
$T : \mathbb{P}\,Net$
$N : \mathbb{P}\,Net$
$I : Net \nrightarrow \mathbb{P}\,Connection$
$O : Net \nrightarrow \mathbb{P}\,Connection$
$HN : Net \nrightarrow Net$
$HP : Place \nrightarrow Net$
$MP : Connection \nrightarrow Place$
$MC : Connection \nrightarrow Connection$
$Top : Net$

---

1 $T \subset N \wedge Top \in N \setminus T \wedge \mathrm{dom}(I) = \mathrm{dom}(O) = N \setminus \{Top\}$
2 $\mathrm{dom}(MP) \cap \mathrm{dom}(MC) = \emptyset$
3 $\mathrm{dom}(MP) \cup \mathrm{dom}(MC) = (\bigcup \mathrm{rng}(I)) \cup (\bigcup \mathrm{rng}(O))$
4 $\forall n_1, n_2 : N \mid n_1 \neq n_2 \bullet I(n_1) \cap I(n_2) = O(n_1) \cap O(n_2) = I(n_1) \cap O(n_2) = \emptyset$
5 $\mathrm{dom}(HN) = N \setminus \{Top\} \wedge \mathrm{rng}(HN) = N \setminus T$
6 $\mathrm{dom}(HP) = P \wedge \mathrm{rng}(HP) = \mathrm{rng}(HN)$
7 $\forall n : \mathrm{dom}(HN) \bullet \exists k : \mathbb{N} \bullet HN^k(n) = Top$
8 $\forall n : \mathrm{dom}(HN) \bullet \forall c : I(n) \cup O(n) \bullet$
  $c \in \mathrm{dom}(MP) \Rightarrow HP(MP(c)) = HN(n) \wedge$
  $c \in \mathrm{dom}(MC) \wedge c \in I(n) \Rightarrow MC(c) \in I(HN(n)) \wedge$
  $c \in \mathrm{dom}(MC) \wedge c \in O(n) \Rightarrow MC(c) \in O(HN(n))$

The first four rules of the above schema are straightforward modifications of rules of the flat net structure. Rules 5, 6 and 7 determine the mappings to the supernets; rule 7 guarantees that in the end everything is mapped to *Top*, so no cycles are possible. The last rule states that if a connection of a subnet is assigned to a place, then both the subnet and the place should be mapped to a common supernet. If it is assigned to a connection, this should be a connection of the supernet.

It is easy to derive a flat net structure from a hierarchical one, by assigning to each connection of a transition the place it is assigned to by the higher level subnets. The next axiomatic description defines a function *nettrans* that maps a hierarchical net structure onto a flat one.

$$
\begin{array}{|l}
\underline{nettrans : HierarchicalNetStructure \rightarrow FlatNetStructure} \\
\forall h : HierarchicalNetStructure \bullet \\
\quad \exists MN : Connection \mathrel{+\!\!\!\rightarrow} Place \bullet \\
\quad\quad \forall t : h.T \bullet \forall c : (h.I)t \cup (h.O)t \bullet \\
\quad\quad\quad c \in \mathrm{dom}(h.MP) \Rightarrow MN(c) = (h.MP)c \;\wedge \\
\quad\quad\quad c \in \mathrm{dom}(h.MC) \Rightarrow MN(c) = MN((h.MC)c) \\
\wedge \\
(nettrans(h)).P = h.P \;\wedge \\
(nettrans(h)).T = h.T \;\wedge \\
(nettrans(h)).I = h.T \lhd h.I \;\wedge \\
(nettrans(h)).O = h.T \lhd h.O \;\wedge \\
(nettrans(h)).M = h.MN
\end{array}
$$

Note that the function *MN* is defined recursively; it is however easy to see that this definition is sound.

In the development process, we start with the definition of a hierarchical net structure. This we transform into a flat net structure. Next we define a state space and finally a net model. In this section we have defined all these concepts, but we did not give a language to *specify* a net model. For the hierarchical net structure we advise a graphical representation such as the one we described and used in the example figures. For the specification of the state space and local state transition functions we advocate Z. This is the topic of the next section.

3 Specification of nets in Z

3.1 Places and transitions

In section 2 we have defined what a net model is. It is constructed from a hierarchical net structure which determines a flat net structure, a state space and local transition functions per net transition. In this section we show how the state space and the local transition function can be specified using Z. In fact we do more: we also specify the net model in Z. Using some examples, we present a method for specification of net models.

Consider the example of a flat net structure given in figure 3. To specify the value domains of the places we use type definitions. Let the types of the four places be T_A, T_B, T_C and T_D. We assume that all types have one common value, denoted by $\perp$ and called

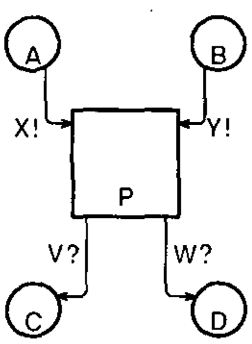

Figure 3: A flat net.

bottom or nil. The places of a net model are specified in one schema called *Places*. In our example we have the following schema for the places

$$\begin{array}{|l}
\textit{Places} \\ \hline
A : T_A \nrightarrow I\!N_1 \\
B : T_B \nrightarrow I\!N_1 \\
C : T_C \nrightarrow I\!N_1 \\
D : T_D \nrightarrow I\!N_1 \\ \hline
\text{true} \\ \hline
\end{array}$$

Note that all places are real bags now and not bags which allow frequency zero as in section 2.

Transitions are specified by three schemas called *front-end*, *body* and *back-end* that have to be combined by the pipe operator $\gg$. This operator has been introduced by [Hayes 87]. Later we will see that we only have to specify the body schema, the other schemas can be derived automatically from the body schema. For transition P in our example we define three schemas P_1, P_2 and P_3 and then $P \mathrel{\hat{=}} P_1 \gg P_2 \gg P_3$.

Schema P_1 describes the selection of tokens: one from place A and one from place B,

$$\begin{array}{|l}
P_1 \\ \hline
\Delta \textit{Places} \\
\Xi(\textit{Places} \backslash (A, B)) \\
X! : T_A \\
Y! : T_B \\ \hline
A \neq \emptyset \land B \neq \emptyset \land \\
X! \in \text{dom}(A) \land Y! \in \text{dom}(B) \land \\
A' = \textit{delete}(X!, A) \land B' = \textit{delete}(Y!, B) \\ \hline
\end{array}$$

Note that both places should contain at least one token. The function *delete* is a generic function with type variable T. It updates a bag of type $T \nrightarrow I\!N$ by deleting an arbitrary element from it.

$$
\begin{array}{|l}
\multicolumn{1}{l}{[T]} \\
\hline
delete : (T \times (T \nrightarrow I\!N)) \nrightarrow (T \nrightarrow I\!N) \\
\hline
\forall t : T \bullet \forall f : T \nrightarrow I\!N \bullet t \in \mathrm{dom}(f) \Rightarrow \\
\quad (t, f) \in \mathrm{dom}(delete) \wedge \\
\quad f(t) > 1 \Rightarrow delete(t, f) = f \oplus \{t \mapsto f(t) - 1\} \wedge \\
\quad f(t) = 1 \Rightarrow delete(t, f) = \{t\} ⩤ f \\
\hline
\end{array}
$$

The following schema of the specification of P is P_3.

$$
\begin{array}{|l}
\multicolumn{1}{l}{P_3} \\
\hline
\Delta Places \\
\Xi(Places \backslash (C, D)) \\
V? : T_C \\
W? : T_D \\
\hline
V? \neq \bot \Rightarrow C' = add(V?, C) \wedge \\
V? = \bot \Rightarrow C' = C \wedge \\
W? \neq \bot \Rightarrow D' = add(W?, D) \wedge \\
W? = \bot \Rightarrow D' = D \\
\hline
\end{array}
$$

This schema uses another generic function, called *add*, that adds one element to a bag.

$$
\begin{array}{|l}
\multicolumn{1}{l}{[T]} \\
\hline
add : (T \times (T \nrightarrow I\!N)) \nrightarrow (T \nrightarrow I\!N) \\
\hline
\forall t : T \bullet \forall f : T \nrightarrow I\!N \bullet \\
\quad t \in \mathrm{dom}(f) \Rightarrow add(t, f) = f \oplus \{t \mapsto f(t) + 1\} \wedge \\
\quad t \notin \mathrm{dom}(f) \Rightarrow add(t, f) = f \oplus \{t \mapsto 1\} \\
\hline
\end{array}
$$

We take special care of the situation that the input has value $\bot$. This case occurs when a transition does not produce a token for a connection.

The schema P_2 specifies the functionality of the transition P. Since we are at this moment not interested in any particular functionality we leave it open by assuming a predicate E. Note that $V!$ or $W!$ may get the value $\bot$, which means that no token is produced.

$$
\begin{array}{|l}
\multicolumn{1}{l}{P_2} \\
\hline
X? : T_A \\
Y? : T_B \\
V! : T_C \\
W! : T_D \\
\hline
E \\
\hline
\end{array}
$$

The final schema P, defined by $P \mathrel{\hat{=}} P_1 \gg P_2 \gg P_3$ joins the three schemas and identifies components with ! and ? and leaves them out of the signature by introducing existential quantors for each of them, as illustrated in the example given in section 3.2. Of course we could have given a direct way to specify P. However, the schemas P_1 and P_3 can be derived from P_2: they do not require any decision from the designer; the designer only has to specify the schema P_2. If we consider $X?$ and $Y?$ as input connections of P and $V!$ and $W!$ as output connections, then we only need to know which places they are assigned to. This information can be derived from the graphical representation of the net structure. Similarly we derive the types of the places by giving the type of the tokens.

So, a full specification of a net model consists of a graphically represented hierarchical net structure, a schema with places and for each transition one schema (the body schema) with input and output connections as components, decorated with ? and ! respectively.

If we generate the front-end and back-end schemas for each transition then we have a complete Z specification with the following operational semantics: if an initial state is given then one of the transitions that is able to fire will do so. Then in the next state again one of the enabled transitions will fire. Note that a transition is enabled if the input places contain enough tokens. In the example this was checked by the first rule of schema P_1. It is possible that like in colored Petri nets there is a precondition based on input values. This can be expressed in the body schema of a transition. In our example the predicate E can have an expression, for instance $X? \neq Y?$.

Note that a specification in this way fits into the metamodel we presented in section 2. The diagram of figure 3 is an instance of the flat net structure schema. The state space is defined by the schema *Places*. To see this recall that in the metamodel a state is of the form $Place \nrightarrow (Value \rightarrow \mathbb{N})$. In the schema *Places* an instance is a tuple or a mapping that assigns to the places A, B, C and D a mapping from respectively T_A, T_B, T_C and T_D to $\mathbb{N}$. Hence T_A, T_B, T_C and T_D are subsets of *Value* and they determine the domains of the places. So we have defined in this way component D of schema *StateSpace*. The local transition function L of schema *NetModel* assigns to each transition a mapping from state to state. The transition schema P for transition P is in fact $L(P)$ since it defines a state transition that obeys the laws of the schema *NetModel*. Hence L is specified by a schema per transition. Finally the operational semantics discussed above and below satisfy the requirements for *Tr* in the schema *NetModel*.

All (serialized) execution paths of a net model can be obtained by constructing all possible lists containing firable transition schemas connected by the ; schema operator. For example for a net consisting of two transitions P and Q, we may get an execution path like

$$\begin{array}{l} P \\ P;Q \\ P;Q;P \\ P;Q;P;P \\ P;Q;P;P;Q \\ \ldots \end{array}$$

3.2 A small example

As a simple example we consider the net structure of figure 4. It consists of one transition P that copies its input token to both output places. The schema representing the places is

```
  Places[T]
 ┌──────────────────────────────
 │ A : T ⇸ ℕ₁
 │ B : T ⇸ ℕ₁
 ├──────────────────────────────
 │ true
 └──────────────────────────────
```

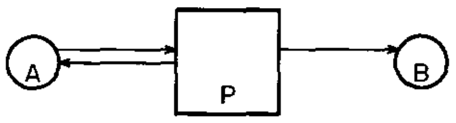

Figure 4: An example.

We have used an arbitrary type T for these places. The front-end, body an back-end schemas for the transition P are, according to the construction in the previous section

$$\begin{array}{|l}
\multicolumn{1}{l}{P_1[T]} \\ \hline
\Delta Places \\
\Xi(Places \backslash (A)) \\
X! : T \\ \hline
A \neq \emptyset \wedge \\
X! \in \mathrm{dom}(A) \wedge \\
A' = delete(X!, A) \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{P_2[T]} \\ \hline
X? : T \\
Y! : T \\
Z! : T \\ \hline
Y! = X? \wedge Z! = X? \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{P_3[T]} \\ \hline
\Delta Places \\
Y? : T \\
Z? : T \\ \hline
A' = add(Y?, A) \wedge \\
B' = add(Z?, B) \\ \hline
\end{array}$$

Then P becomes, after removing most of the existential quantors

$$\begin{array}{|l}
\multicolumn{1}{l}{P} \\ \hline
\Delta Places \\ \hline
A' = A \wedge \exists Y : T \bullet B' = add(Y, B) \\ \hline
\end{array}$$

3.3 Stores

In many applications it is useful to have a local state for a net transition. This can be modeled by a special place that always contains one token and that is both an input and an output place for the transition. We call such a place a *store*. (For stores we use a graphical representation differing from normal places.) Two transitions may have access to the same store; however only one at a time since there is only one token in the place representing the store. This avoids concurrent update problems. Usually a store has a rather complex type. It is not necessary to define the store as a bag since there is always one token; this will simplify the specifications considerably. So we may specify the type of a store just by the type of the token. In data base applications this token usually has a rather complex type that may be defined by several auxiliary schemas. As an example we construct for a store A a database type.

$$\begin{array}{|l}
\multicolumn{1}{l}{\quad Person} \\ \hline
name : NAME \\
address : NODE\ ADDRESS \\
age : \mathbb{N} \\ \hline
age \geq 0 \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\quad Car} \\ \hline
license : \mathbb{N} \\
kind : KIND \\
horsepower : \mathbb{N} \\
weigth : \mathbb{N} \\ \hline
horsepower * 10 \geq weight \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\quad Trip} \\ \hline
driver : NAME \\
license : \mathbb{N} \\
starttime : \mathbb{N} \\
destination : PLACE \\ \hline
\mathrm{true} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\quad A} \\ \hline
person : \mathbb{P}(Person) \\
car : \mathbb{P}(Car) \\
trip : \mathbb{P}(Trip) \\ \hline
\forall t : trip \bullet t.driver \in \{x : person \bullet x.name\} \wedge \\
\qquad\qquad t.license \in \{x : car \bullet x.license\} \\ \hline
\end{array}$$

Note that the condition in schema A is a referential integrity constraint for the database.

3.4 An alternative place construction

We could have considered an alternative for the place construction. Up to here we followed the line of section 2. However, this leaves open how to choose tokens from the places. An alternative could be to replace the bags by queues. In fact this is the way we have implemented it in ExSpect. We have to replace the schema for the places by

$$\begin{array}{|l}
\multicolumn{1}{l}{\quad Places} \\ \hline
A : \mathrm{seq} T_A \\
B : \mathrm{seq} T_B \\
C : \mathrm{seq} T_C \\
D : \mathrm{seq} T_D \\ \hline
\mathrm{true} \\ \hline
\end{array}$$

to account for the change of bag into queue. The front-end and back-end schemas for figure 3 now become equal to

$$\begin{array}{|l}
\multicolumn{1}{l}{P_1} \\ \hline
\Delta \mathit{Places} \\
\Xi(\mathit{Places} \backslash (A, B)) \\
X! : T_A \\
Y! : T_B \\ \hline
A \neq [\,] \wedge B \neq [\,] \wedge \\
X! = \mathrm{head}(A) \wedge Y! = \mathrm{head}(B) \wedge \\
A' = \mathrm{tail}(A) \wedge B' = \mathrm{tail}(B) \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{P_3} \\ \hline
\Delta \mathit{Places} \\
\Xi(\mathit{Places} \backslash (C, D)) \\
V? : T_C \\
W? : T_D \\ \hline
V? \neq \perp \Rightarrow C' = C \frown [V?] \wedge \\
V? = \perp \Rightarrow C' = C \wedge \\
W? \neq \perp \Rightarrow D' = D \frown [W?] \wedge \\
W? = \perp \Rightarrow D' = D \\ \hline
\end{array}$$

Note that P_2 does not have to be changed.

4 A distributed telephone index

To illustrate the concepts of the previous sections we present a simple distributed specification of a reactive system. It is a distributed telephone index: it receives questions for phone numbers and gives answers.

Let *NAME* be the set of all names and let *NUMBER* be the set of all valid telephone numbers. At the top level we may specify the index system (as seen from the environment of one node) as a schema *TI*. This is possible since we are dealing with a reactive system: it only produces tokens as a reaction upon an incoming token of the environment.

$$\begin{array}{|l}
\multicolumn{1}{l}{TI_2} \\ \hline
X? : \mathit{NAME} \\
Y! : \mathit{NUMBER} \\
\mathit{index} : \mathit{NAME} \rightarrow \mathit{NUMBER} \\ \hline
Y! = \mathit{index}(X?) \\ \hline
\end{array}$$

The corresponding net is given in figure 5. In this net we have modeled the environment

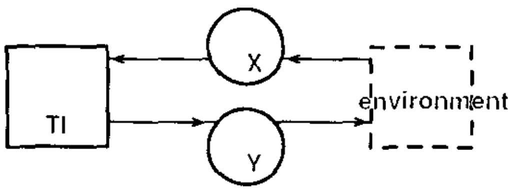

Figure 5: The toplevel of the index system.

as a subnet which we do not specify any further.

The next step is to decompose the net *TI* into several subnets *LTI*, each representing a local node. In a local node questions for numbers arrive from the local environment; if possible such a question is answered directly, if not a message is sent into a network of remote telephone indices. After a cycle through this network, an answer will arrive. The decomposition of the net *TI* is given in figure 6.

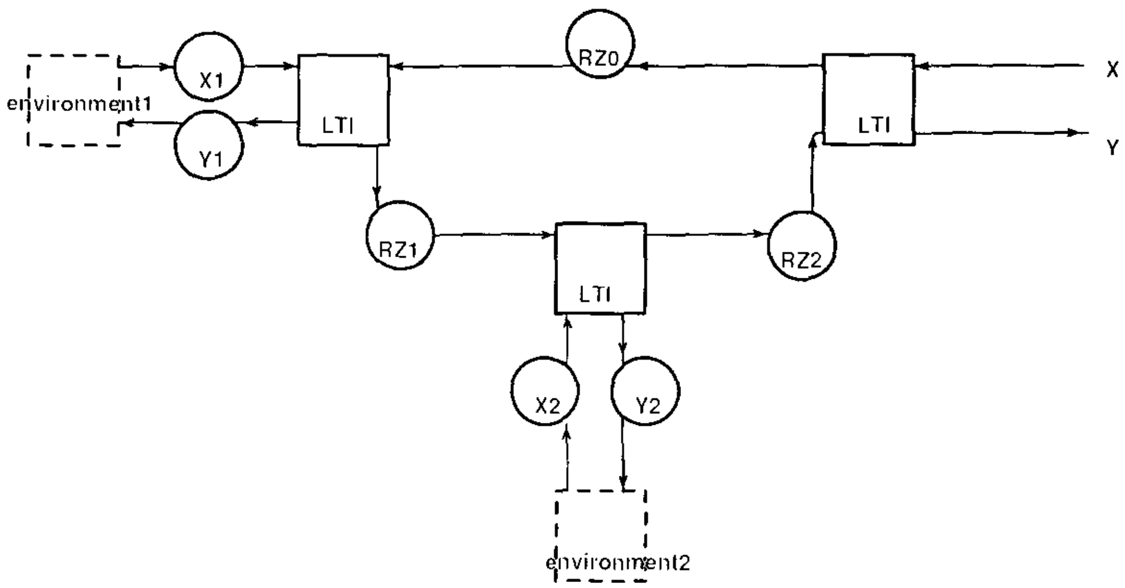

Figure 6: The decomposition of the toplevel *TI*.

We use the following schema to represent the type of a message

mes

$name : NAME$ $number : NUMBER$ $address : NODE\,ADDRESS$

where *NODEADDRESS* is the set of all node addresses. The *address* field denotes the node address from which the message originated and *name* denotes the name for which a number should be found.

We define the network of figure 7 as the decomposition of a local index system *LTI*. Each local index system i has stores containing its local index of names and numbers and its node address.

$Stores_i$

$address_i : NODE\,ADDRESS$ $index_i : NAME \rightarrowtail\!\!\!\!\!\!+ \; NUMBER$
true

The places we need are the following

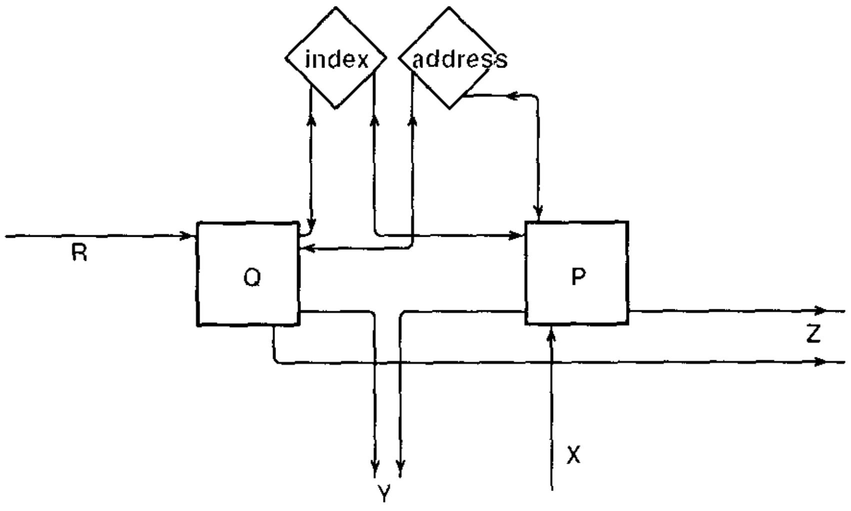

Figure 7: A local index system *LTI*.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Places}} \\
\hline
R : mes \twoheadrightarrow \mathbb{N}_1 \\
X : NAME \twoheadrightarrow \mathbb{N}_1 \\
Y : NUMBER \twoheadrightarrow \mathbb{N}_1 \\
Z : mes \twoheadrightarrow \mathbb{N}_1 \\
\hline
\text{true} \\
\hline
\end{array}$$

Questions enter the local index system at X, answers are given at Y and the incoming and outgoing network connections are supplied by R and Z. The specifications of the bodies of the two transitions P and Q in the local index system are given below.

$$\begin{array}{|l}
\multicolumn{1}{l}{P_2} \\
\hline
\Xi Stores_i \\
X? : NAME \\
Y! : NUMBER \\
Z! : mes \\
\hline
X? \in \operatorname{dom}(index_i) \wedge Y! = index_i(X?) \wedge Z! = \bot \\
\vee \\
X? \notin \operatorname{dom}(index_i) \wedge Y! = \bot \wedge Z!.name = X? \wedge Z!.address = address_i \\
\hline
\end{array}$$

In reaction to a question on $X?$, transition P searches the local index and gives an answer on $Y!$ if the name is present. Otherwise it puts a message for information on $Z!$. The (local) index and address are never changed, so we have used $\Xi Stores_i$.

Q_2

$$\begin{array}{|l}
\Xi Stores_i \\
R? : mes \\
Z! : mes \\
Y! : NUMBER \\
\hline
R?.address = address_i \wedge Y! = R?.number \\
\vee \\
R?.address \neq address_i \wedge R?.name \notin \mathrm{dom}(index_i) \wedge Z! = R? \\
\vee \\
R?.address \neq address_i \wedge R?.name \in \mathrm{dom}(index_i) \wedge \\
\quad Z!.name = R?.name \wedge Z!.address = R?.address \wedge Z!.number = index_i(R?.name)
\end{array}$$

Transition Q takes an incoming *NAME,NUMBER,NODEADDRESS* triple from $R?$ and sends the number to $Y!$ if the address matches the local address. This happens when a message sent by this node has traversed the whole network. Otherwise it searches the local index for the name and forwards the triple to the next node via $Z!$.

5 Conclusions

In this paper we have shown how Z can be used (a) to specify the metamodel of a hierarchical colored Petri net and (b) to specify the transitions in a specific colored Petri net. We have seen that Z is very well suited for such a specification. However, a few small comments remain. It is not possible to specify constants as parameters to schemas (types are possible). Therefore in our example in section 4 we have used a subscript i. Furthermore when one wants an executable specification, one needs to restrict the predicates in the body schemas of the transitions.

A method for constructing specifications of distributed systems starts by building a hierarchical net model in a graphical way and continues by specifying the primitive transitions by means of Z schemas. As a next design step, the Z schemas should be transformed into functions in order to get an executable specification as in ExSpect.

References

[Aalst 90] W.M.P.van der Alst and A.W.Waltmans, Modeling Logistic Systems with ExSpect, in: H.G.Sol, K.M.van Hee (eds.), Dynamic Modeling of Information Systems, North-Holland, 1991.

[Albrecht 89] K.Albrecht, K.Jensen and R.M.Shapiro, Design/CPN: A tool package supporting the use of Colored Petri Nets, Petri Net Newsletter 32, 1989.

[Diepen 90] M.J.van Diepen and K.M.van Hee, A Formal Semantics for Z and the link between Z and the Relational Algebra, in: D.Bjørner, C.A.R.Hoare, H.Langmaack (eds.), VDM'90, VDM and Z - Formal Methods in Software Development, Lecture Notes in Computer Science 428, Springer Verlag, 1990.

[Giovanni 90] R.Di Giovanni and P.L.Iachini, HOOD and Z for the Development of Complex Software Systems, in: D.Bjørner, C.A.R.Hoare, H.Langmaack (eds.), VDM'90, VDM and Z - Formal Methods in Software Development, Lecture Notes in Computer Science 428, Springer Verlag, 1990.

[Hayes 87] I.Hayes (ed.), Specification Case Studies, Prentice Hall, 1987.

[Hee 89] K.M.van Hee, L.J.Somers and M.Voorhoeve, Executable Specifications for Distributed Information Systems, in: E.D.Falkenberg, P.Lindgreen (eds.), Information system concepts: an in-depth analysis, North-Holland, 1989.

[Hee 91] K.M.van Hee and P.A.C.Verkoulen, Integration of a Data Model and Petri Nets, in: Proceedings 12th International Conference on Application and Theory of Petri Nets, Århus, Denmark, 1991.

[Jensen 91] K.Jensen, Colored Petri Nets: A High Level Language for System Design and Analysis, in: G.Rozenberg (ed), Advances in Petri Nets 1990, Lecture Notes in Computer Science 483, Springer Verlag, 1991.

[Reisig 87] W.Reisig, Petri Nets in Software Engineering, in: W.Brauer, W.Reisig, G.Rozenberg (eds.), Petri Nets: Applications and Relationships to other Models of Concurrency, in: Lecture Notes in Computer Science 255, Springer Verlag, 1987.

[Spivey 89] J.M.Spivey, The Z Notation: A Reference Manual, Prentice Hall, 1989.

[Yourdon 89] E.Yourdon, Modern structured analysis, Prentice-Hall 1989.

90/1	W.P.de Roever- H.Barringer- C.Courcoubetis-D.Gabbay R.Gerth-B.Jonsson-A.Pnueli M.Reed-J.Sifakis-J.Vytopil P.Wolper	Formal methods and tools for the development of distributed and real time systems, p. 17.
90/2	K.M. van Hee P.M.P. Rambags	Dynamic process creation in high-level Petri nets, pp. 19.
90/3	R. Gerth	Foundations of Compositional Program Refinement - safety properties - , p. 38.
90/4	A. Peeters	Decomposition of delay-insensitive circuits, p. 25.
90/5	J.A. Brzozowski J.C. Ebergen	On the delay-sensitivity of gate networks, p. 23.
90/6	A.J.J.M. Marcelis	Typed inference systems : a reference document, p. 17.
90/7	A.J.J.M. Marcelis	A logic for one-pass, one-attributed grammars, p. 14.
90/8	M.B. Josephs	Receptive Process Theory, p. 16.
90/9	A.T.M. Aerts P.M.E. De Bra K.M. van Hee	Combining the functional and the relational model, p. 15.
90/10	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).
90/11	P. America F.S. de Boer	A proof system for process creation, p. 84.
90/12	P.America F.S. de Boer	A proof theory for a sequential version of POOL, p. 110.
90/13	K.R. Apt F.S. de Boer E.R. Olderog	Proving termination of Parallel Programs, p. 7.
90/14	F.S. de Boer	A proof system for the language POOL, p. 70.
90/15	F.S. de Boer	Compositionality in the temporal logic of concurrent systems, p. 17.
90/16	F.S. de Boer C. Palamidessi	A fully abstract model for concurrent logic languages, p. p. 23.
90/17	F.S. de Boer C. Palamidessi	On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.

90/18	J.Coenen E.v.d.Sluis E.v.d.Velden	Design and implementation aspects of remote procedure calls, p. 15.
90/19	M.M. de Brouwer P.A.C. Verkoulen	Two Case Studies in ExSpect, p. 24.
90/20	M.Rem	The Nature of Delay-Insensitive Computing, p.18.
90/21	K.M. van Hee P.A.C. Verkoulen	Data, Process and Behaviour Modelling in an integrated specification framework, p. 37.
91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.

91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.
91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.
91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. .
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.