I261T001011 . 261
T001011d.261

Smart Engineering System Design, Vol. 00, pp. 1–17
Reprints available directly from the publisher
Photocopying permitted by license only

High Level Petri Nets and Rule Based Systems for Discrete Event System Modelling

DUMITRU DAN BURDESCU[a], MARIUS BREZOVAN[a] and DAN B. MARGHITU[b,*]

[a]*University of Craiova, Romania;* [b]*Auburn University, AL 36849, USA*

(Received 7 June 2000; In final form 8 December 2000)

This paper is threefold structured: the first part presents *High Level Petri Nets*, a formalism to specify concurrent and discrete event systems. The definition of *High Level Petri Nets* is made by specification of both formal syntax and dynamic semantics. To do this, a formal definition of composed data types is made, in a similar way to the structured data types of programming languages. The second part of the paper presents a way for implementing the dynamic semantics of High Level Petri Nets by using a modified version of RETE algorithm. A connection between rule-based systems and High Level Petri Nets is presented, and it is sown that for every Petri net, a rule-based semantic equivalent system exists. Third, a rule based language for system modelling is described and an example of using *High Level Petri Nets* for modelling a manufacturing system is presented.

Keywords: Petri nets; Rule-based systems; System modelling; Discrete event systems; Algebraical specifications

1. INTRODUCTION

In this paper, a new formalism called *High Level Petri Nets* is defined, which try to unify the main classes of Petri nets with tokens having data types: Predicate/Transition nets, algebraical nets and colored Petri nets [7, 9–11, 19, 20, 27]. Predicate/Transition nets have a connection to rule-based systems, colored Petri nets use the data type concept, and algebrical nets have a rigorous mathematical support offered by algebraical specifications.

There are different ways to introduce annotations in Petri nets. In this paper *algebraical specifications* are used, which are defined by Ehrig and Mahr [4]. Another mathematical concept used in High Level Petri Nets definition is the notion of *multiset*, which is used to define the marking of a net, and the arc annotations.

The cartesian product of the form $A_1 \times A_2 \times \cdots \times A_k$ from the Predicate/Transition nets is substituted with composed data types, each component of a such structured type corresponds

*Corresponding author. Tel.: (334) 844-3335, Fax: (334) 844-3307, e-mail: marghitu@eng.auburn.edu

to a set A_i of the product. In this way, each place of a net corresponds to a data type, and this gives a compatibility with the colored Petri nets. Also, the tuples of constants associated to places correspond to simple or structured constants, which represent the marking of these places. Another important difference is that the formal sums are substituted with the multisets.

An algorithm for semantics implementation of Petri nets has to generate a subset of reachability graph of the net starting with the initial marking. A such algorithm simulates the evolution of a system described by a corresponding Petri net. During the past years, various papers about Petri net implementation have been published. The centralised implementation is based on the concept of token-player [3, 26] and it is used in the case when a Petri net describes the internal behaviour of a task rather than a global behaviour of a set of tasks. When the Petri net describes a set of communicating tasks, a distributed implementation can be used [2, 3].

In this paper, a form of a centralised Petri net implementation based on a RETE algorithm is proposed. To do this, a connection between rule-based systems and High Level Petri Nets is presented: for every Petri net, a semantic equivalent rule-based system exists.

A suggestive method for discrete event system modelling is proposed in the last section of the paper. This method uses production rules, because their similarities with the High Level Petri Nets. An example of modelling a flexible manufacturing system using this method is shown.

2. COMPOSED DATA TYPES

In this section, the composed data types are defined by using algebraic specifications [4].

Using a Σ-presentation can specify a data type. Intuitively, a *simple data type* is defined by a Σ-presentation (S, F, EQ) where the set of sorts S has a single element: $\text{card}(S) = 1$. In the case of *composed data type* the set S has more than one element: $\text{card}(S) > 1$.

The set of the constants associated to a data type contains the *base terms* of that type.

DEFINITION 1 Let $P = (S, F, EQ)$ be a Σ-presentation.

(1) The set of all *base terms* over the signature Σ, denoted by $((T_\Sigma)_s)_{s \in S}$, (or T_Σ when there is not confusion in context), is the smallest set defined inductively as follow:

   (a) $\forall f: \lambda \rightarrow s \in F : f \in (T_\Sigma)_s$
   (b) $\forall (t_1, \ldots, t_n) \in (T_\Sigma)_{s1} \times \cdots \times (T_\Sigma)_{sn}$, $\forall f: s_1 \ldots s_n \rightarrow s : f(t_1, \ldots, t_n) \in (T_\Sigma)_s$.

(2) The *set of the constants* of the type associated to presentation P is the set T_Σ: $CON_S = T_\Sigma$.

Let $\Sigma = (S, F)$ be a signature and X an S-indexed family of sets of variables. Since both the set of terms over signature Σ and variables from X (denoted by $T_{\Sigma,X}$) and the set of base terms over signature Σ can be organised as free algebra, the notion of variable assignment may be used without a specified Σ-algebra.

DEFINITION 2 Let $P = (S, F, EQ)$ be a Σ-presentation with a signature $\Sigma = (S, F)$ and X an S-indexed family of sets of variables. A *base assignment* is an S-morphism from X to T_Σ. The *set of base assignments* in X is denoted by $ASS_S(X)$.

DEFINITION 3 Let $Sp = (S, F, X, EQ)$ be an algebraic specification.

(1) An *instance* of Sp is a tuple (S, F, X, EQ, σ), where $\sigma \in ASS_S(X)$.
(2) The *set of instances* of specification Sp is denoted by Inst(Sp), and is defined as: $\text{Inst}(Sp) = \{(S, F, X, EQ, \sigma) | \sigma \in ASS_S(X)\}$.

The set of instances of all specifications is denoted by INST.

Remarks

(1) Let VAR denoting the set of all variables.
(2) Let PRES denoting the set of all Σ-presentations, and SPEC the set all algebraical specifications.
(3) Let $\mathrm{SORT_B} \subseteq \mathrm{SORT}$ be the set of names of all simple types and let $\mathrm{SORT_C}$ ($\mathrm{SORT_C} \subseteq \mathrm{SORT}$, $\mathrm{SORT_C} \cap \mathrm{SORT_B} = \Phi$) be the set of names of all composed types.
(4) Let β and ρ be two maps for the specification of the simple, and the composed types:

$$\beta: \mathrm{SORT_B} \to \mathrm{PRES},$$
$$\rho: \mathrm{SORT_C} \to \mathrm{SPEC}.$$

For a correct definition of composed types it is required that these types are not auto-refereed. A function *tp* can used to specify this condition:

$$\begin{aligned} &\mathrm{tp}: \mathrm{SORT} \to \wp(\mathrm{SORT}),\\ &\quad \mathrm{tp}(s) = S, \quad \text{where } S \subseteq \mathrm{SORT}, \text{ and:}\\ &\quad \beta(s) = (S, F, EQ), \quad \text{if } s \in \mathrm{SORT_B},\\ &\quad \rho(s) = (S, F, X, EQ), \quad \text{if } s \in \mathrm{SORT_c}. \end{aligned}$$

This function may be extended for $S \subseteq \mathrm{SORT}$ by:

$$\mathrm{tp}(S) = \cup_{s \in S} \mathrm{tp}(s).$$

Now, the following recursive function can be defined. For an arbitrary $S \subseteq \mathrm{SORT}$ and an integer n, the function tp^n is defined as follows:

$$\begin{aligned} &\mathrm{tp}^0(S) = \mathrm{tp}(S),\\ &\mathrm{tp}^n(S) = \mathrm{tp}(\mathrm{tp}^{n-1}(S)), \quad \text{for } n > 0. \end{aligned}$$

DEFINITION 4

(1) A *simple data type* is a Σ-presentation:

$$P = (S, F, EQ),$$

for which a sort $s \in \mathrm{SORT_B}$ exists so that $\beta(s) = P$ and $\mathrm{card}(S) = 1$.

(2) A *composed data type structured by name* is an algebraical specification:

$$Sp = (S, F, X, EQ),$$

for which a sort $s \in \mathrm{SORT_C}$ exists so that $\rho(s) = Sp$ and $\forall n \in N$, $s \notin \mathrm{tp}^n(s)$.

For simplicity, a composed date type $Sp = \rho(s)$ is denoted by its name s.

With previous defined functions, if $Sp = (S, F, X, EQ)$ defines a composed data type, the condition for non-infinite recursive definition of Sp can be written as:

$$\forall n \in N, \quad s \notin \mathrm{tp}^n(s).$$

DEFINITION 5 Let $s \in \mathrm{SORT_C}$ be a composed data type and $\rho(s) = (S, F, X, EQ)$.

(1) A *structured constant* c, of the type s is an instance of the specification $\rho(s)$:

$$c = (S, F, X, EQ, \sigma),$$

where $\sigma \in \mathrm{ASS_S}(X)$.
(2) The *set of all constants* of the type s is the set of all instances of the specification $\rho(s)$:

$$\begin{aligned} \mathrm{CON_s} &= \mathrm{Inst}(\rho(s))\\ &= \{(S, F, X, EQ, \sigma) | \rho(s)\\ &\qquad = (S, F, X, EQ) \text{ and } \sigma \in \mathrm{ASS_S}(X)\}. \end{aligned}$$

Let CONST denoting the set of all simple and structured constants.

The reference of a component from a composed element is made by the reference of the associated variable. The reference operation of a component is usual called *selection*.

DEFINITION 6 Let $Sp = (S, F, X, EQ)$ be a specification. The selection operator is a map:

$$(\cdot) : \mathrm{VAR} \times \mathrm{VAR} \to \mathrm{VAR},$$
$$\cdot\,(x, y) = x \cdot y = y,$$

so that, if $x \in X_s$ and $\rho(s) = (S^1, F^1, X^1, EQ^1)$, then a sort $s1 \in SORT$ exists, and $y \in X^1_{s1}$.

The syntactic construction $x \cdot y$ means that y represent the component selected from the variable x. If $\sigma \in ASS_S(X)$ and $(S^1, F^1, X^1, EQ^1, \sigma^1) \in Inst(\rho(s))$, then by extension:

$$\sigma(x \cdot y) = \sigma^1(y).$$

Because for every signature $\Sigma = (S, F)$ and for a family X of S-indexed variables, an algebra of terms over signature Σ and variables from X is possible to be constructed, it is necessary to develop this construction for the case when the selector operator appears. Each selection $x \cdot y$ is considered to be a new variable y from a new specification $\rho(s)$.

DEFINITION 7 Let $s \in SORT_C$ be a composed type and $\rho(s) = (S, F, X, EQ)$. An *elementary transformation* of s is a map:

$$\begin{aligned}
&\alpha: SPEC \to SPEC,\\
&\alpha(\rho(s)) = (S^1, F^1, X^1, EQ^1), \quad \text{where:}\\
&\quad S^1 = S \cup S^2, F^1 = F \cup F^2, X^1 = X \cup X^2,\\
&\quad EQ^1 = EQ \cup EQ^2, \quad \text{where:}\\
&\quad S^2 = \cup_{p(s)} S^3, F^2 = \cup_{p(s)} F^3,\\
&\quad EQ^2 = \cup_{p(s)} EQ^3, X^2 = \cup_{p(s)} \mu(X^3), \quad \text{where:}\\
&\quad p(s) = (s \in S \cap SORT_C)\\
&\qquad \wedge (\rho(s) = (S^3, F^3, X^3, EQ^3)) \quad \text{and}\\
&\mu(X^3) \subseteq VAR \text{ is an isomorphic}\\
&\qquad \text{transformation of the set } X^3\\
&\qquad \text{that redefines the variables from } X^3\\
&\qquad \text{to generate disjoint sets of}\\
&\qquad \text{variables.}
\end{aligned}$$

Using a single transformation, it is possible that the obtained specification contains also the composed subtypes. In this case an elementary transformation must be applied again.

DEFINITION 8 Let α be an elementary transformation. The *transitive closure* of α is defined as follows:

(1) let α^n be a recursive map defined as follows: $\alpha^n = \alpha \cdot \alpha^{n-1}$; $\alpha^0 = \alpha$;
(2) let n be the smallest non-negative integer so that: $\alpha^n(S, F, X, EQ) = (S^1, F^1, X^1, EQ^1)$, and $S^1 \cap SORT_C = \Phi$;
(3) the transitive closure of α is denoted by α^+ and defined as: $\alpha^+ = \alpha^n$.

THEOREM 1 *For every specification* $Sp = (S, F, X, EQ)$, *a unique specification* Sp^1 *exists with the property that:* $Sp^1 = (S^1, F^1, X^1, EQ^1)$, *and* $S^1 \cap SORT_C = \Phi$.

Proof Immediately from the previous definitions and concepts. A homomorphism:

$$\mu: SPEC \to SPEC$$

is defined so that:

$$\begin{aligned}
&\mu = Id \text{ (identity function)}, \quad \text{if } S \cap SORT_C = \Phi,\\
&\mu = \alpha^+, \quad \text{if } S \cap SORT_C \neq \Phi.
\end{aligned}$$

The transitive closure exists from the condition that the composed data types are not infinite recursive.

For every specification $Sp = (S, F, X, EQ)$, another specification Sp^1 that contains only elementary types can be uniquely defined:

$$Sp^1 = (S^1, F^1, X^1, EQ^1);$$

Moreover, for this specification Sp^1, the associated algebra of elementary terms $T_{\Sigma 1, X1}$ can be constructed.

THEOREM 2 *For every algebraic specification, there exist a unique associated algebra of elementary terms, denoted by* T_{Sp}.

Proof Let $Sp^1 = \mu(Sp)$. If $Sp^1 = (S, F, X, EQ)$, then $T_{Sp} = T_{\Sigma, X}$.

3. HIGH LEVEL PETRI NETS DEFINITION

This section formally defines the High Level Petri Nets formalism through unification of Predicate/Transition Nets, Coloured Petri Nets and Algebraic Nets.

DEFINITION 9 Given an algebraic specification, $Sp = (S, F, X, EQ)$, a *High Level Petri Net* is a triplet:

$HLPN = (N, Ins, M_0)$, where:

(1) $N = (P, T, A)$ is an elementary Petri net:

- P and T are disjoint sets of places and transitions: $P \cap T = \Phi$;
- A is a finite set of arcs: $A \subseteq P \times T \cup T \times P$;

(2) $Ins = (\varphi, L, R)$ is the *net annotation*:

- $\varphi: P \to S$ is a map which associates every place to a sort of S;
- $L: A \to (T_{\Sigma,X})_{MS}$, is a map which associates each arc to a multiset of terms, compatible with the type of adjacent place of arc:

$$\forall a \in A,\ a = (p,t) \text{ or } a = (t,p) \Rightarrow \forall v \in \mathrm{Supp}(L(a)),\ v \in (T_{\Sigma,X})_{\varphi(p);}$$

- $R: T \to (T_{Sp})_{bool}$, is a map, which associate each transition to a logical formula called the *selector* of transition;

(3) $M_0: P \to (\cup_{s \in S} CON_s)_{MS}$, represents the *initial marking* of the net and it is a map which associates each place to a multiset of constants compatible with the type of place:

$$\forall p \in P, \quad \forall c \in \mathrm{Supp}(M_0(p)) \Rightarrow c \in CON_{\varphi(p)}.$$

Remarks

(1) The multisets associated to every arc *a* of the net must contain only constants and variables belonging to the type of the place connected to the *a*;
(2) The selectors associated to transitions are boolean terms;
(3) $(T_{sp})_{bool}$ represents an algebra of boolean terms associated to $T_{\Sigma,X}$, which may contain selector operators;
(4) CON_s represents the set of structured constants of the type *s*;
(5) K_{MS} represents the set of all multiset sets over the set K.

The *semantics* of a High Level Petri Net is defined by using the notions of *marking*, *enabling* and *firing*.

DEFINITION 10 A *marking* M of a High Level Petri Net is a map:

$$M: P \to (\cup_{s \in S} CON_s)_{MS},$$

which associates each place to a multiset of tokens, with a sort compatible to the sort of the place:

$$\forall p \in P, \forall c \in \mathrm{Supp}(M(p)) \Rightarrow c \in CON_{\varphi(p)}.$$

The initial marking and the marking of a High Level Petri Net represent a multiset over composed constants.

DEFINITION 11 Let HLPN be a High Level Petri Net, M a marking of HLPN, t a transition and α an occurrence mode of t. The transition t is called to be *M-enabled* with occurrence α, and it is denoted $M[t: \alpha\rangle$, if the following relations hold:

(1) $\forall p \in {}^\circ t \Rightarrow \mu_\alpha(L(p,t)) \leq M(p)$;
(2) $\forall p \in t^\circ \Rightarrow \mu_\alpha(L(t,p)) \cdot M(p) = \Phi$;
(3) $\mu_\alpha(R(t)) = \mathrm{true}$.

Remark μ_α represents an interpretation in the Σ-algebra associated with the algebraical specification of the High Level Petri Net, and the operator $\cdot$ represents the operation of multiset multiplication.

The occurrence of an event leads to a new marking.

DEFINITION 12 Let HLPN be a High Level Petri Net and t be a transition M-enabled with occurrence α. The *firing* of transition t leads to a following marking M^1, denoted $M[t: \alpha\rangle\ M^1$ and defined by:

- $\forall p \in t^\circ \cup {}^\circ t$,
  $M^1(p) = M(p) - \mu_\alpha(L(p, t)) + \mu_\alpha(L(t, p))$;
- $\forall p \notin t^\circ \cup {}^\circ t$, $M^1(p) = M(p)$.

4. HIGH LEVEL PETRI NETS AND RULE-BASED SYSTEMS

The structure of a rule-based system contains three mainly elements:

- a *fact base*, called the working memory of the system, where the known facts at different moments are stored;
- a *rule base* which contains the rules used to infer new facts;
- an *inference engine* which selects some applicable rules to infer new facts.

The rules from the rule base are syntactic structures of the form:

If ⟨conditions⟩ ***then*** ⟨actions⟩

The conditions represent the rule *antecedent* and they are patterns that are tested for the rule activation. If the conditions match with the facts from the fact base, the rule can fire, and the actions representing the *consequent* of the rule are performed.

The pattern-matching process is performed during the inference process, and the variables appearing into the patterns are bound to some constants of the fact base.

There are two important classes of inference algorithms: forward chaining algorithms, and backward chaining algorithms. Because Petri nets are used mainly for concurrent and distributed system specification and for that systems forward chaining inference is usually used, in the following only the forward chaining inference algorithms are considered.

The main structure of a forward chaining algorithm is presented in procedure *Infer*:

```
procedure Infer (rule-base, fact-base)
  selected-rules←Select(rule-base, fact-base)
  while selected-rules ≠ Φ do
    rule←SolveConicts (selected-rules)
    ApplyRule (rule)
    selected-rules←Select (rule-base, fact-base)
  od
end
```

The function *Select* selects a set of rules from the rule base whose antecedent matches with the facts from the fact base. The function *Match* determines this matching process. One of the rules selected will be fired, which is determined by function *SolveConflicts*. The procedure *ApplyRule* fires a rule by performing the actions of its consequent.

An improved version of this algorithm was described by Forgy [6], which allows to increase the speed of pattern-matching process. This algorithm is called RETE, and it tries to avoid the iterations over facts of the fact base by storing for each pattern a list of the matched rule antecedents.

Similarities between Predicate/Transition nets and rule based systems have been established [15, 17, 21], and High Level Petri Nets preserve this property. One can show that for every High Level Petri Net a semantic equivalent rule based system exists.

To do this, an appropriate syntax of the production rule language, closed to the High Level Petri Nets definition is chosen.

⟨rule⟩ ::= ***if*** ⟨antecedent⟩ ***then*** ⟨consequent⟩ ***end***

⟨antecedent⟩ ::= λ|⟨pattern⟩ {∧⟨pattern⟩}* [∧⟨test⟩]

⟨pattern⟩ ::= ⟨ident⟩ (⟨multiset⟩)

⟨test⟩ ::= ***test*** (⟨boolean expression⟩)

⟨multiset⟩ ::= ⟨integer⟩* ⟨ident⟩ {, ⟨integer⟩* ⟨ident⟩ }*

⟨consequent⟩ ::= ⟨action⟩ {∧⟨action⟩}*
⟨action⟩ ::= ⟨append⟩|⟨delete⟩
⟨append⟩ ::= ***add*** (⟨ident⟩, '{'⟨multiset⟩'}')
⟨delete⟩ ::= ***del*** (⟨ident⟩, '{'⟨multiset⟩'}')
⟨fact⟩ ::= '{'⟨multiset of constants⟩'}'
⟨multiset of constants⟩ ::= ⟨integer⟩* ⟨const⟩ {, ⟨integer⟩*⟨const⟩}*

The transformation of a High Level Petri Net into a rule based system is made through the following steps:

(1) Rule base construction: one production rule is associated to each transition of the net, as follows:
   - a pattern is associated to each input place of the transition; the parameters of the pattern represents the expression L(a) of the arc connecting the place to the transition;
   - a test condition is associated to the selector of the transition, specifying its boolean expression;
   - a delete action is associated to each output place of the transition, in a similar way to the pattern association;
   - an append action is associated to each output place of the transition, by specifying the place name and the expression L(a) of the arc connecting the place to the transition.

(2) Construction of initial fact base that contains all multisets of the initial marking:

$$F = \bigcup_{p \in P} (M_0(p)).$$

In other words, a fact represents a multiset of constants, which is compatible with the type of its associated place.

PROPOSITION 1 *For every High Level Petri Net, a semantic equivalent rule based system exists.*

Proof A High Level Petri Net can be transformed into a knowledge base according to the Steps (1) and (2), having the property that a biunivocal correspondence exists between the net transitions and system rules. The following remarks state that the rules fired by inference algorithm in procedure *Infer* represents a subset of the reachability graph of the initial net:

(a) Pattern-matching operation, whose effect is to bind the antecedent variables to appropriate values, is made by matching the multiset of variables (belonging the input arcs of the transition) against the multiset of constants representing the marking of those places.
(b) Antecedent evaluation operation represents the verification of the two conditions from the definition of transition enabling. Through the pattern-matching operation, the occurrence mode of the transition is realised.
(c) Performing the consequent actions of a rule represents the operations performed when the correspondent transition fires; this means that rule firing is semantic equivalent to the correspondent transition firing.

It follows that inference algorithms can be used to implement the semantics of the High Level Petri Nets.

5. RETE-BASED ALGORITHMS FOR HIGH LEVEL PETRI NETS SEMANTICS IMPLEMENTATION

The most proposals for Petri nets semantics implementation uses token-player algorithms [13, 24, 25]. A *token-player* simulation algorithm for Petri nets can be simple described by an infinite loop, at each step the enabled transitions are determined and one enabled transition is fired. The main disadvantage is the fact that at each iteration all transition of the top level net must be tested [25]. An optimal alternative uses the launch places [24]. Every transition has assigned an input place, which is tested for enabling: if this place has a non-empty marking, the corresponding transition is possible to be enabled; else the transition is certainly non-enabled.

The algorithm presented in this paper is based on the RETE algorithm for rule-based systems, and avoid the testing of all transitions for enabling determination. The tokens are propagated in the RETE graph, as much as possible; when tokens reach an output node, they activate the transition associated to that node. In this way, the presented algorithm is faster than a token-player algorithm.

In place to translate a net into a production system [3], the RETE algorithm works directly with a High Level Petri Net. The general structure of a sorting net adapted to the High Level Petri Nets is presented in Figure 1.

The inputs for the sorting net correspond to the places of the associated High Level Petri Net, and the output places to the transitions of the net. The sorting net contains two subnets: the *distribution net* and the *test net*.

The distribution net is composed by a number of trees equal to the number of places of the High Level Petri Net, each tree having two levels: the root and the leaves. To simplify the test net, if a place is an input place for more transitions, this place must be duplicated. As a consequence, the leaves from a distribution tree are duplications of its associated place.

The test tree contains one test tree for each transition. The root of the tree is an output node for the sorting net, and the leaves represent outputs of the distribution net.

The construction of the sorting net starts with the test subnet, because the distribution subnet is a simple list of lists, which can be easy constructed if the test subnet is constructed.

In order to simplify the construction operation, one can observe some elements:

(1) Even though the sorting net is viewed as a directed graph, however it represents in fact a set of disjoint trees: n-ary trees in the distribution subnet, and binary trees in the test subnet; the coupling of these subnets is made by using the leaf nodes. The attributes *left*, *right* and *down* are used to specify the links to the neighbour nodes.
(2) As in RETE graphs, there are two types of nodes in the test net: (a) *one input nodes*, called also *input nodes* and representing the leaf nodes of the test trees, and (b) *two input nodes*, which can be either *internal nodes* or *external nodes* (representing the roots of the test trees).
(3) Two attributes are used for the input/output

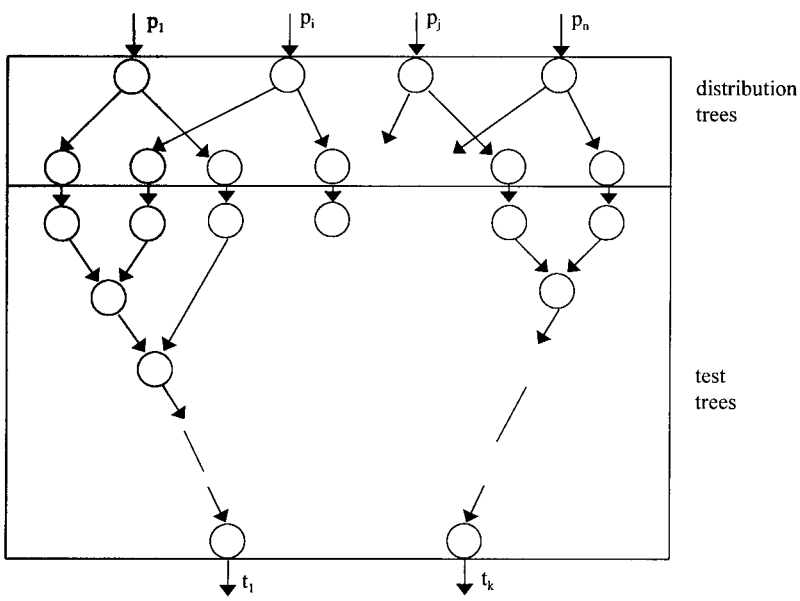

FIGURE 1 Structure of a sorting net.

nodes of the sorting to specify the connections to the environment: *place* for input nodes and *transition* for output nodes.

Denoting by $p_1, p_2, \ldots, p_k$ the input places for a transition t, and by p_{1m} the two input node obtained from the ascendants p_1 and p_m, the order for nodes generating (in the test tree for transition t) is: p_1, p_2, p_{12}, p_3, p_{123}, p_4, $p_{1234}, \ldots, p_n$, $p_{12\ldots n}$. The root of the tree is $p_{12\ldots n}$, for which $transition(p_{12\ldots n}) = t$. The leaf nodes are: p_1, $p_2, \ldots, p_n$, for which $place(p_i) = p_i$, $i = 1, 2, \ldots, n$.

The algorithm that creates the test net is presented in function *TestNetCreation*, where *lq* is an output parameter, and *node_list* is a list of nodes:

```
node_list function TestNetCreation
(lq)
    initq (lq)
    forall t∈T do
      lq←TestTreeCreation (t, lq)
    od
    return lq
  end
```

Each test tree is created by using the function *Test TreeCreation*, where *lq* is an output parameter. The distribution net is generated having the list *lq* of the leaf nodes of the test net as input parameter, and the list *lp* of the root nodes of the distribution trees as output parameter.

```
node-list function Distribution-
NetCreation (lq)
    initq (lp)
    forall p∈P do
      n←DistributionTreeCreation
(p, lq)
      addq (lp, n)
    od
    return lp
  end
```

A distribution tree represents in fact the root node (whose *place* attribute points to the associated place of the High Level Petri Net) and a list of its descendants (indicated by the attribute *down*).

To manage both the tokens from the place markings, and the variable assignment of the occurrence mode of a transition, the nodes from the sorting net have two additionally attributes: *tokens*, which represents a multiset of constants, and *variables*, which represents a list of pairs of the form (variable-value).

The tokens propagation is distinct in the two subnets: in the distribution net they are copied in the leaf nodes, while in the test net they descends from a level to the next level, if it is possible. When tokens reach an output node, they activate the transition associated to that node.

The algorithm to add a multiset of tokens in a place of a High Level Petri Net is specified in the procedure *AddTokensRete*, which has two input parameters: the place and the multiset of tokens:

```
procedure AddTokenRete (p, e)
  q←Search (lp, p)
  forall n∈down (q) do
    tokens (n) ←tokens (n)∪e
    tokens (place (n))←
      Match (p, L (p, transition
(place (n))), var)
    variables (place (n))←var
    Propagate (place (n))
  od
end
```

The function *Search* determines the input node of the sorting net associated to the place p. The procedure $Match(p, L(p,t), var)$ performs the matching process of the tokens from the marking $M(p)$ with the expression $L(p,t)$.

The procedure *Propagate* propagates the multiset of tokens (as a result of the matching process) in the test net:

```
procedure Propagate (q)
  term←false
  if ntype (q)≠0 then
    q←down (q)
    while ¬term∧(ntype (q)≠0) do
      term←Perform (q)
      q←down (q)
    od
```

```
    end
```

The variable *term* is used to determine when an internal node founds a discordance between the assignments of a variable in two distinct places (by using the function *Perform*):

```
    logic function Perform (q)
      if (left (q)≠λ)∧(right (q)≠λ)
then
        if DifferentValues (variables
(left (q)),
            variables (right (q))) then
          return false


      tokens (q)←tokens (left (q))∪
tokens (right (q))
      variables (q)←variables (left
(q))∪variables (right (q))
      if ntype (q) =0 then
        variables (q)←variables (q)∪
            AssignOut (transition (q))
        α(transition (q))←variables
(q)
        addq (enabled, transition (q)))

      return true
    end
```

In the case when the function *Perform* works on an output node of the sorting net, and the matching is not failed, the transition associated to this node is added to the list of the enabled transitions. For simplicity one can assume that the list *enabled* is a global variable of the algorithm.

To delete a multiset of constants from a place marking is similar to the adding operation, unless the fact that the tokens are not propagated in the sorting net.

One can observe that the transition enabling operation is made in the procedure *AddTokens-Rete* by using the function *Perform*. The firing procedure calls these two procedures:

```
    procedure FiringTransitionRete (t,
a)
      forall p∈°t do
        DelTokensRete (p, Eval (L(p,
t)))
      od
      forall p∈t° do
        AddTokensReteT (p, Eval (L(t,
p)))
      od
    end
```

The main algorithm first generates the sorting net associated with the High Level Petri Net, initialises the sorting net, and calls the procedure *ReteCycle*:

```
    procedure ReteControl
      InitRete
      ReteCycle
    end
    procedure InitRete
      lq←TestNetCreation
      lp←DistributionNetCreation (lq)
      initq (enabled)
      forall p∈P do
        if M0 (p)≠Φ then
          AddTokensRete (p, M0 (p))
        forall
      od
      end
    procedure ReteCycle
      while enabled≠Φ do
        t1←delq (enabled)
        FiringTransitionRete (t1)
      od
    end
```

7. PRODUCTION RULE BASED LANGUAGE FOR SYSTEM MODELLING

Petri nets and production rules are two important formalisms used to discrete event systems specifi-

cation. Petri nets represent a rigorous mathematical founded formalism, while production rules a well suitable formalism to describe the behaviour of these systems. Because of similitude between production rules and High Level Petri Nets, production rules can be used as a description language for High Level Petri Nets.

The syntax of a production rule language has been presented in the Section 4. Each transition corresponds to a production rule. The conditions of the rule antecedent can be patterns or test expressions. A pattern can be matched with the marking associated to the input places of the transition, and a test expression represents the selector of that transition. The actions associated to the rule consequent can be the adding/deleting of token multisets to/from place markings, or variable assignments.

Enabling of a transition corresponds to triggering the associated rule. In the case of conjunctive clauses, the antecedent of a rule is true if all its conditions are true. A pattern is true if it matches with the tokens of the marking of the specified place. If a rule is triggered, it can fire and the actions implied by consequent conclusions are taken.

The syntax of the production rule language can be simplified. Because the semantics of the High Level Petri Nets, the multisets of variables associated to the input arcs of a transition must be twice written: in a pattern from the rule antecedent for rule triggering determination, and in a delete action from the rule consequent to remove the tokens when the transition fires. For example, the transition t_a from the Figure 2 has the following associated rule:

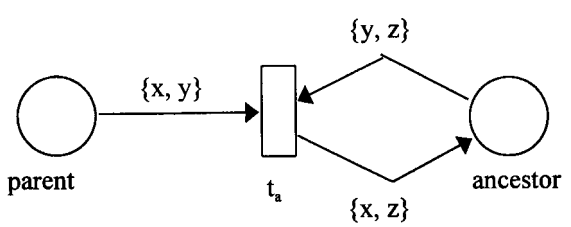

FIGURE 2 A Petri net associated to a production rule.

$$t_a : \textit{if}\ \text{parent(x,y)} \wedge \text{ancestor(y,z)}\ \textit{then}\ \text{del(parent(x,y))} \wedge \text{del(ancestor(y,z))} \wedge \text{add(ancestor(x,z))}\ \textit{end}$$

To simplify the syntax, the delete actions can be omitted: when a transition fires, all the tokens bound to variables during the pattern-matching process are implicitly removed from the corresponding input places of the transition. Moreover, for an append action the keyword *add* can be omitted. The previous rule can be written as follows:

$$t_a : \textit{if}\ \text{parent(x,y)} \wedge \text{ancestor(y,z)}\ \textit{then}\ \text{ancestor(x,z)}\ \textit{end}$$

Another useful action is the bind operation: a variable used in a rule can be bound to a value after the evaluation of an expression. A simple syntax for a production rule system can be described as follows:

⟨rule⟩ ::= ***if*** ⟨antecedent⟩ ***then*** ⟨consequent⟩ ***end***
⟨antecedent⟩ ::= λ|⟨pattern⟩ {∧⟨pattern⟩}* [∧⟨test⟩]
⟨pattern⟩ ::= ⟨ident⟩ (⟨multiset⟩)
⟨test⟩ ::= ***test*** (⟨boolean expression⟩)
⟨multiset⟩ ::= ⟨integer⟩* ⟨ident⟩ {, ⟨integer⟩* ⟨ident⟩}*
⟨consequent⟩ ::= ⟨action⟩ {∧⟨action⟩}*
⟨action⟩ ::= ⟨append⟩|⟨bind⟩
⟨append⟩ ::= ⟨ident⟩ (⟨multiset⟩)
⟨bind⟩ ::= ⟨variable⟩←⟨expression⟩

The activity of system modelling can be made in the following steps:

(1) determination of the system activities;
(2) determination of necessary resources;
(3) association of activities and resources to appropriate data types;
(4) for each activity, two transitions are associated: one transition for activity starting and one transition for activity ending;

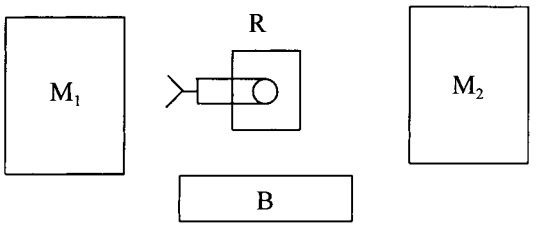

FIGURE 3 Flexible manufacturing cell.

(5) all pairs of transitions associated to consecutive activities (the end of the first activity with the start of the second one) are merged;
(6) for each of remained transitions, a production rule is associated.

The following example illustrates the previous modelling method. A flexible manufacturing cell is sketched in Figure 3. The system consists of two different machines (M_1 and M_2), a robot R, and a buffer B. The following process plan is considered:

(a) only one part type are processed;
(b) each final product requires two operations, op1 on the machine M_1, and op2 on the machine M_2.
(c) op1 must be accomplished before op2;
(d) after the finishing of op1, the robot unload the machine M_1 and stores the intermediate part on the buffer;
(e) the robot load the machine M_2 with an intermediate part from the buffer;
(f) when the operation op2 ends, the robot unload the machine M_2, defixes the finished part, and returns the palette to the machine M_1.

The High Level Petri Net associated to the manufacturing system is constructed as follows:

1. The required activities are: the loading of M_1 (place RL_1), the processing on machine M_1 (place M_1P), the unloading of M_1 (place RU_1), the storage of an intermediate part on the buffer (place BS), the loading of M_2 (place RL_2), the processing on machine M_2 (place M_2P), and the unloading of M_2 (place RU_2). The resources considered are: the machines M_1 and M_2, the robot R, the buffer B, and the fixed parts.
2. The activity sequencing is the following:
   (a) RL_1: R takes a part on the input part storage, fixtures the part, and load the machine M_1;
   (b) M_1P: M_1 processes the part;
   (c) RU_1: R unload M_1;
   (d) BS: the buffering activity (R stores a part on the buffer B);
   (e) RL_2: R load M_2 with an intermediate part from B;
   (f) M_2P: M_2 processes the part;
   (g) RU_2: R unload M_2, defixtures the finished part, and returns the palette to M_1.
3. The places associated to the resources required to be available at the start of each activity are the following:
   (a) for the activity RL_1, the machine M_1, the robot R, and an available palette are required; it results the places M_1A, RA, and PA;
   (b) for the activity M_1P there are not required resources;
   (c) for the activity RU_1, the robot R is required (the place RA is already created);
   (d) for the activity BS, an available place on buffer is required; this is modelled by the place BA;
   (e) for activity RL_2, R and M_2 are required; a new place (M_2A) is created (RA is already created);
   (f) for activity M_2P there are not required resources;
   (g) for activity RU_2, the robot R is required; the place RA is already created.
4. The following places are created at the end of each activity:
   (a) at the end of activity RL_1, the robot R becomes available; RA is already created;
   (b) at the end of activity RU_1, the machine M_1, and the robot R become available; the places M_1A ßi RA are already created;

(c) at the end of activity BS, a place in buffer B becomes available; the place BA is already created;
(d) at the end of activity RL_2, the robot R becomes available; RA is already created;
(e) at the end of activity RU_2 the machine M_2, the robot R, and a palette become available; the places M_1A, RA, and PA are already created.

5. The following data types are associated to places:

(a) each of the places: $\{RL_1, M_1P, RU_1, BS, RL_2, M_2P, RU_2\}$ specify an activity in progress, and are associated to a simple data type: action = {in-progress};
(b) the places describing the availability of resources, have the following data types:

– for M_1A and M_2A: machine = struct {
code: string;
operations: {op1, op2, op3, op4}
}
– for RA: robot = struct{
code: string;
. . . . . .
}
– for PA and BA: available {av}

6. The initial marking specifies the initial state of the system, denoted by the following facts: there are five pallets available, both the machines M_1 and M_2 are available, the robot is available, and the buffer have four places available:

$$M_0(M_1A) = \{\{M1, \{op1, op3\}\}\},$$

$$M_0(M_2A) = \{\{M2, \{op2, op4\}\}\},$$

$$M_0(RA) = \{\{R\}\},$$

$$M_0(BA) = \{4^*av\}$$

$$M_0(PA) = \{5^*av\}$$

7. The following rules describe the transition for activities starting:

t_1: ***if*** there is an available palette, and robot is available, and M_1 is available ***then*** R loads M_1 ***end***
t_2: ***if*** M_1 is loaded ***then*** start process on M_1, and robot is available ***end***
t_3: ***if*** robot is available, and end process on M_2 ***then*** start unloading M_1 ***end***
t_4: ***if*** M_1 unloaded, and available a place in buffer
then M_1A is available, and stores the part to buffer, and robot is available ***end***
t_5: ***if*** the part stored in buffer, and robot is

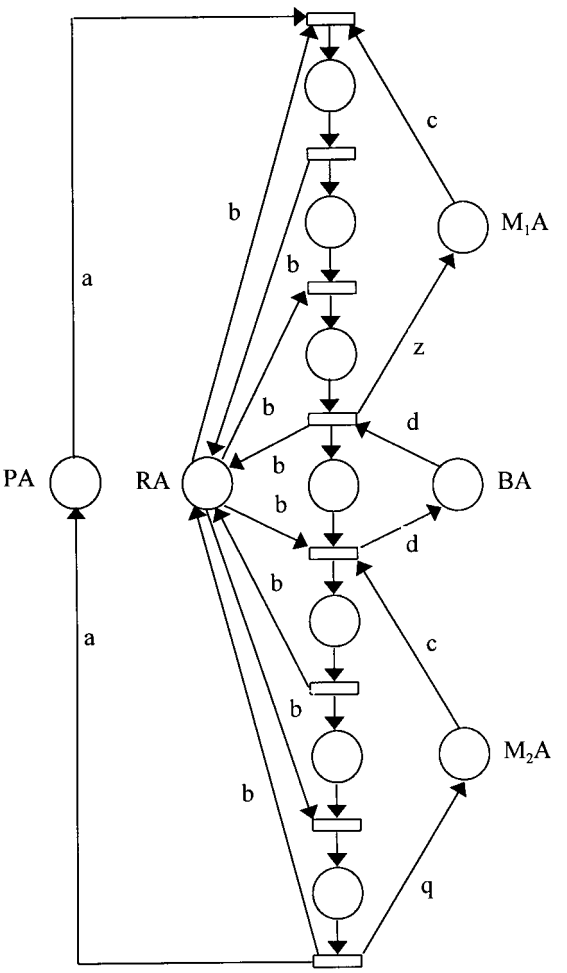

FIGURE 4 The Petri net associated to the rule base.

available
then start loading M_2, and available a place in buffer ***end***

t_6: ***if*** M_2 is loaded ***then*** start process on M_2 and robot is available ***end***

t_7: ***if*** end process on M_2, and robot is available ***then*** start unloading M_2 ***end***

t_8: ***if*** M_2 unloaded ***then*** M_2 is available, and robot is available, and palette is available ***end***

Using variables and places above described, the rules can be formalised as follows:

t_1: ***if*** PA(a) and RA(b) and M_1A(c) ***then*** x←in-progress and RL_1(x) ***end***

t_2: ***if*** RL_1(x) ***then*** M_1P(x) and b←{R} and RA(b) ***end***

t_3: ***if*** RA(b) and M_1P(y) ***then*** RU_1(y) ***end***

t_4: ***if*** RU_1(z) and BA(d) ***then*** M_1A(z) and BS(z) and b←{R} and RA(b) ***end***

t_5: ***if*** BS(u) and RA(b) ***then*** RL_2(u) and d←av and BA(d) ***end***

t_6: ***if*** RL_2(v) ***then*** M_2P(v) and b←{R} and RA(b) ***end***

t_7: ***if*** M_2P(p) and RA(b) ***then*** RU_2(p) ***end***

t_8: ***if*** RU_2(q) ***then*** M_2A(q) and b←{R} and RA(b) and a←av and PA(a) ***end***

8. The associated net is described in Figure 4.

8. CONCLUSIONS

This paper has three main goals:

(a) to construct a formalism called *High Level Petri Nets*, used to specify discrete event systems;
(b) to describe a method for semantics implementation of these nets by using an algorithm based on the inference methods of the rule-based systems;
(c) to define a language for production rules based on High Level Petri Nets, which can be used for discrete event system modelling.

The formalism of High Level Petri Nets is viewed as a general union of the most important high level Petri nets formalisms: Predicate/Transition Nets, Coloured Petri Nets and Algebraic Nets.

The main reason for introducing High Level Petri Nets is the necessity to formally define the simple and complex data types in a way similarly to programming languages. This formalism may be used in implementation of this net classes and in development of an programming environment. Some features of main high level Petri net classes are presented underneath:

(a) Predicate/Transition nets use relational structures and first order predicate logic language to specifies the tokens types and annotation of arcs; these definitions do not allow a direct implementation of nets;
(b) Coloured Petri nets use the notion of data type, which is not formal and rigorous defined;
(c) Algebraical nets use algebraical specification to define abstract data types; although they are generally, however they do not allow direct implementation using a programming language and they do not allow the specification of usual operations for transition selectors;
(d) High Level Petri Nets use algebraical specification for a formal and rigorous definition of nets, and they contain the data type notion in a way similarly to programming languages; this fact allows a simple implementation of nets.

A method of implementing the semantics of High Level Petri Nets is presented, based on the inference algorithms of the rule based systems. The algorithm presented in this paper is based on the RETE algorithm, and avoid the testing of all transitions for enabling determination.

The proposed algorithms are algorithms for simulation, and the High Level Petri Nets can be used for discrete event system modelling. An important problem in the system modelling is the language used to specify the Petri net, which describe de modelled system. The most proposals

in system modelling with Petri nets, both text oriented and graphic oriented, describe the Petri net configuration, rather than the modelled system [3, 5, 13, 16]. The language proposed in this paper uses the production rules formalism, having two main advantages: (a) it describes the working of the modelled system, and (b) it allows to generate automatically the corresponding High Level Petri Net.

References

[1] Bauman, R. and Turano, T. A. (1986). Production based language simulation of Petri nets, *Simulation*, **47**, 191 – 198.

[2] Bruno, G. and Marchetto, G. (1986). Process-translatable Petri nets for the rapid prototyping of process control systems, *IEEEE Transactions on Software Engineering*, Vol. SE-12(2), pp. 346 – 357.

[3] Colom, J. M., Silva, M. and Villarroel, J. L. (1986). On software implementation of Petri nets and coloured Petri nets using high-level concurrent languages, *European Workshop on Application and Theory of Petri nets*, pp. 207 – 241.

[4] Ehrig, H. and Mahr, B. (1985). Fundamentals of Algebraic Specifications, *Vol. 6 of EATCS Monograph in Theoretical Computer Science* (Springer-Verlag).

[5] Fleischack, H. and Weber, A., Rule-based programming, Predicate/Transition nets and the modeling of the procedures and flexible manufacturing systems, *Proc. of the 10th International Conference on Applications and Theory of Petri Nets* (June, 1989).

[6] Forgy, C. (1982). RETE: A Fast Algorithm for Many Pattern/Many Objects, *Artificial Intelligence*, **19**, 17 – 37.

[7] Genrich, H. J. (1987). Predicate/Transition Nets, *Lecture Notes in Computer Science*, **254**, 207 – 247.

[8] He, X. (1996). A Formal Definition of Hierarchical Predicate Transition Nets, *Lecture Notes in Computer Science*, **1091**, 212 – 229.

[9] Jensen, K. (1986). Coloured Petri Nets, *Lecture Notes in Computer Science*, **254**, 248 – 299.

[10] Jensen, K. (1992). Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol. 1: Basic Concepts, *EATCS Monographs on Theoretical Computer Science*.

[11] Jensen, K. and Rozenberg, G. Eds. (1991). *High-level Petri Nets* (Springer-Verlag New York).

[12] Martinez, J. and Silva, M. (1985). A language for the description Concurrent Systems Modelled by Coloured Petri Nets: Application to the Control of Flexible Manufacturing Systems, *Chapter 8 in Languages for Automation* (Plenum Publishing Co.), pp. 369 – 388.

[13] Martinez, J., Muro, P. R. and Silva, M. (1987). Modelling, Validation and Software Implementation of Production Systems Using High Level Petri Nets, *Proc. of the IEEE International Conference in Robotics and Automation*, pp. 1180 – 1185.

[14] Martinez, J. M., Muro, P. R., Silva, M., Smith, S. F. and Villarroel, J. L. (1988). Merging artificial intelligence techniques and Petri nets for real-time scheduling and control of production systems, *Proc. of the 12th IMACS World Congress on Scientific Computation*, **3**, 528 – 531.

[15] Murata, T. and Zhang, D. (1988). A Predicate-Transition net model for parallel interpretation of logic programs, *IEEE Transactions on Software Engineering*, **14**(1).

[16] Nelson, R. A., Haibt, L. M. and Sheridan, P. B. (1983). Casting Petri Nets into Programs, *IEEE Transactions on Software Engineering*, **9**(5), 590 – 602.

[17] Peterka, G. and Murata, T. (1989). Proof procedure and answer extraction in Petri net model of logic programs, *IEEE Transactions on Software Engineering*, **15**(2).

[18] Ramaswamy, S. and Valavanis, K. P., Hierarchical Time-Extended Petri Nets (H-EPNs) Based Error Identification and Recovery for Hierarchical Systems, *IEEE Transactions on Systems, Man and Cybernetics* (Feb, 1996).

[19] Reisig, W. (1991). Petri nets and algebraic specifications, *Theoretical Computer Science*, **80**, 1 – 34.

[20] Reisig, W. and Vautherin, J. (1987). An algebraic approach to high level Petri nets, *Proc. of the Workshop on Applications and Theory of Petri Nets*, Zaragoza, pp. 51 – 72.

[21] Sahroui, A., Atabakhche, H., Courvoisier, M. and Valette, R. (1987). Joining Petri Nets and Knowledge Based Systems for monitoring purposes, *Proc. of the IEEE International Conference on Robotics and Automation*, pp. 1861 – 1867.

[22] Sibertin-Blanc, C., High-level Petri nets with data structures, *Proc. of the 6th European Workshop on Application and Theory of Petri Nets* (June, 1985).

[23] Sifakis, J. (1977). Use of Petri nets for performance evaluation, *Measuring, Modeling and Evaluating Computer Systems* (North-Holland), pp. 75 – 93.

[24] Taubner, D. (1988). On the Implementation of Petri Nets, *Lecture Notes in Computer Science*, **340**.

[25] Valette, R. (1995). Petri nets for control and monitoring: specification, verification, implementation, *Proc. of the workshop "Analysis and Design of Event-Driven Operations in Process Systems"*, London.

[26] Valette, R. and Bako, B. (1991). Software Implementation of Petri Nets and Compilation Rule-based Systems, *Lecture Notes in Computer Science*, **524**, 296 – 316.

[27] van Hee, K. M. and Verkoulen, P. A. C. (1991). Integration of a Data Model and High-Level Petri Nets, *Lecture Notes in Computer Science*.

[28] Zisman, M. D. (1978). Use of production systems for modelling asynchronous concurrent processes, *Pattern Directed Inference Systems* (Academic Press), pp. 53 – 68.

APPENDIX A

Algebraical Specifications

An *abstract data type* is a data structure together with a set of operations, which generate an algebra. In the following an universe **U** is considered, which include the following disjoint sets: the set of all functions denoted by **FUNC**, the set of all sort names denoted by **SORT**, and the set

of all variable names denoted by **VAR**.

DEFINITION A.1 A signature is a pair (S, F), where S is a set of sorts $S \subseteq SORT$ and $F \subseteq FUNC$ is a family of sets indexed in $S^* \times S$:

$$F = (F_{w,s})_{w \in S^*, s \in S}.$$

An element $f \in F_{s1 \ldots sn,s}$ can also be denoted by $f: s_1 \ldots s_n \rightarrow s$.

DEFINITION A.2 Let $\Sigma = (S, F)$ be a signature. A ***Σ-algebra*** is a pair (A, F^A), where $A = (A_s)_{s \in S}$ is a family of S-indexed sets, and:

$$F^A = (f^A_{s1 \ldots sn,s})_{f:s1 \ldots sn \rightarrow s \in F}$$

$f^A_{s1 \ldots sn,s}$ is a function $f^A_{s1 \ldots sn,s} : A_{s1} \times \cdots \times A_{sn} \rightarrow A_s$

DEFINITION A.3 Let $\Sigma = (S, F)$ be signature, A and B be two Σ-algebras. A ***Σ-homomorfism*** $\mu: A \rightarrow B$ is a family of S-indexed maps $(\mu_s: A_s \rightarrow B_s)_{s \in S}$ for which:

$$\mu_s(f^A_{s1 \ldots sn,s}(a_1, \ldots, a_n)) = f^B_{s1 \ldots sn,s}(\mu_s(a_1), \ldots, \mu_s(a_n)).$$

The properties of a data type operations are specified by using axioms which are pairs of well formed terms over a signature.

DEFINITION A.4 Let $\Sigma = (S, F)$ be a signature and X be a family of S-indexed set of variables $X \subseteq VAR$. The set of all terms over the signature Σ and variables from X is denoted $(T_{\Sigma X})_S)_{s \in S}$ or $T_{\Sigma,X}$ and represents the smallest set defined as follows:

(1) $\forall x \in X_s$: $x \in (T_{\Sigma,X})_s$
(2) $\forall f: \lambda \rightarrow s \in F$: $f \in (T_{\Sigma,X})_s$
(3) $\forall (t_1, \ldots, t_n) \in (T_{\Sigma,X})_{s1} \times \cdots \times (T_{\Sigma,X})_{sn}$,
$\forall f$: $S_1 \ldots S_n \rightarrow S$: $f(t_1, \ldots, t_n) \in (T_{\Sigma,X})_s$.

In previous definition λ represents empty sequence and the operator f has arity 0 (*i.e.*, $f: \lambda \rightarrow s$) and it is the set of all constant symbols of type $s \in S$ (*i.e.*, the set A_s). For this reason an usually notation for A_s is CON_s.

Two important notions are the assignment and the evaluation.

DEFINITION A.5 Let A be a Σ-algebra and $X \subseteq VAR$ be a family of S-indexed set of variables. An assignment σ for the variables of X is a S-morfism from X to A.

DEFINITION A.6 Let Σ be a signature, A be a Σ-algebra and σ be an assignment for the variables of X in A. An evaluation in A is a Σ-morfism μ_σ from $T_{\Sigma,X}$ to A so that:

(1) $\mu_\sigma(\text{const}) = \text{const}_A$;
(2) $\mu_\sigma(x) = \sigma(x)$, where $x \in X$;
(3) $\mu_\sigma(f^A(t_1, \ldots, t_n)) = f^A(\mu_\sigma(t_1), \ldots, \mu_\sigma(t_n))$.

DEFINITION A.7 A ***Σ-equation*** is a triple (X, t_1, t_2) where X is a set of S-indexed variables and t_1, $t_2 \in (T_{\Sigma,X})_s$, $s \in S$ are terms of same type. Usually a Σ-equation is denoted by $t_1 = t_2$.

DEFINITION A.8 An algebraical specification is a tuple:

$$Sp = (S, F, X, EQ)$$

where:

$\Sigma = (S, F)$ is a signature X a set of S-indexed variables and EQ is a set of Σ-equations.

APPENDIX B

Multisets

DEFINITION B.1 Let K be a finite nonempty set.

(1) A multiset m over K is a function $m: K \rightarrow N$
(2) $m(k) \in N$ represents the number of appearances of $k \in K$ in the multiset m.
(3) An element $k \in K$ belongs to m iff $m(k) \neq 0$.
(4) The set of all sets over K is denoted $\mathbf{K_{MS}}$:

$$K_{MS} = \cup_{L \in \wp(K)} MS(L).$$

Two usually notations for a multiset m over K are:

(1) $m = \Sigma_{a \in K}\ m(a)a$;
(2) $m = \{k_1 m(k_1), k_2 m(k_2), \ldots, k_n m(k_n)\}$, where $K = \{k_1, k_2, \ldots, k_n\}$;

DEFINITION B.2 If m is a multiset over K, the set K is called support of m and is denoted $K = \text{Supp}(m)$.

For multisets the following operation are defined:

DEFINITION B.3 Let K a finite nonempty set m, m_1, $m_2 \in K_{MS}$, $c \in N$.

(1) The sum of the multisets m_1, and m_2 is denoted by $m_1 + m_2$ and defined as:

$$(m_1 + m_2)(k) = m_1(k) + m_2(k), \quad \forall k \in K;$$

(2) The product of the multisets m_1 and m_2 is denoted by $m_1{}^*\ m_2$ and defined as:

$$(m_1^* m_2)(k) = m_1(k)^* m_2(k), \quad \forall k \in K;$$

(3) The scalar multiplication of the multiset m by constant c is denoted by $c \cdot m$ and defined as:

$$(c \cdot m)(k) = c^* m(k), \quad \forall k \in K;$$

(4) Non-equality relation of the multisets m_1 and m_2 is denoted by $m_1 \neq m_2$ and defined as:

$$m_1 \neq m_2 \Leftrightarrow \exists k \in K\colon\ m_1(k) \neq m_2(k);$$

(5) Less than or equal relation of the multisets m_1 and m_2 is denoted by $m_1 \leq m_2$ and defined as:

$$m_1 \leq m_2 \Leftrightarrow \exists k \in K\colon\ m_1(k) \leq m_2(k);$$

(6) The order relations, $>$, $\geq$, $=$ are defined analogously;
(7) If $m_1 \leq m_2$ then it may be defined the difference $m_1 - m_2$ as follows:

$$(m_1 - m2)(k) = m_1(k) - m_2(k), \quad \forall k \in K.$$

DEFINITION B.4 Let A be a Σ-algebra with the signature $\Sigma = (S, F)$, X be a family of S-indexed set of variables, $T_{\Sigma,X}$ be the set of all terms over the signature Σ and variables from X σ an assignment of variables from X in A, μ_σ be an evaluation of terms from $T_{\Sigma,X}$ A_{MS} be the set of all multisets over A and $(T_{\Sigma,X})_{MS}$ be the set of all multisets of terms from $T_{\Sigma,X}$.

An interpretation in $(T_{\Sigma,X})_{MS}$ is a map ν from $(T_{\Sigma,X})_{MS}$ A_{MS} defined as follows:

$$\nu(\{k_1 t_1, k_2 t_2, \ldots, k_n t_n\}) = \{k_1 \mu_\sigma(t_1),\ k_2 \mu_\sigma(t_2), \ldots, k_n \mu_\sigma(t_n)\}.$$

The interpretation of a multiset of terms is also a multiset.