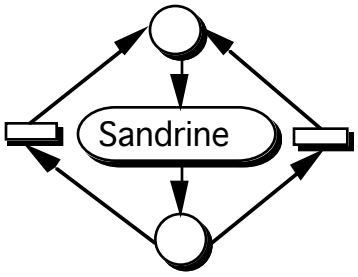


SANDRINE :
an Analysis System for the Declaration of AMI-Nets

X. Bonnaire, C. Dutheillet, S. Haddad

Laboratoire MASI /Institut Blaise Pascal
Université Paris 6 / CNRS URA 818
4 place Jussieu
75 252 Paris Cedex 05

RESUME

La description d'un réseau de Petri de haut-niveau comporte, outre la liste usuelle des places et des transitions, un certain nombre d'informations spécifiques et présentées sous forme textuelle, telles que les domaines de couleur ou les expressions des fonctions sur les arcs. Cette information n'est exploitable par des outils de calcul qu'à condition d'être décrite suivant une syntaxe préalablement définie. SANDRINE est un analyseur dont le rôle est d'effectuer un maximum de vérifications sur la syntaxe et la sémantique d'un modèle en vue de pouvoir étudier celui-ci avec les outils intégrés dans l'atelier AMI. Les modèles reconnus par l'analyseur correspondent au formalisme des AMI-Nets intégré dans l'atelier. Ce formalisme est dérivé du modèle théorique des réseaux bien formés dont la puissance d'expression est identique à celle des réseaux colorés généraux. Il est utilisé actuellement par les outils de prototypage, de calcul de flots et de réductions, et d'analyse de propriétés de type conflit.

ABSTRACT

Besides the usual list of places and transitions, the description of a high-level Petri net model includes some specific textual information, such as the color domains or the expression of valuation functions. This information can be used by software tools only if it is described according to a well defined syntax. SANDRINE is an analyzer that aims at performing a maximum of checks on the syntax and the semantics of a model so that this model can be studied with the tools integrated in the AMI CASE. The models accepted by the analyzer correspond to the AMI-Net formalism defined in the CASE. This formalism is derived from the theoretical model of Well-Formed Nets whose modelling power is identical to that of general colored nets. At present, it is used by prototyping tools, flow computation and reduction tools, together with tools analyzing conflict-like properties.

1. INTRODUCTION

SANDRINE is a new tool in the AMI CASE. It aims at a syntactic and semantic analysis of a model, in order to perform as many checks as possible and to provide an enriched description of the model.

The models analyzed by SANDRINE must obey a well-defined syntax, which is however easily extendable for new classes of models. The model description must be in accordance with a grammar that applies both to the declaration and to the valuation of the arcs of the net.

At present, the grammar of SANDRINE defines a model that we call AMI nets, which is derived from the theoretical model of Well-Formed nets. We present below the main features of that model.

1.1. AMI Nets

The AMI net model is derived from the theoretical model of Well-Formed Nets. It is a high-level net model that includes, besides the graphical features of an ordinary Petri net - places, transitions and arcs - textual information that can be classified in three types :

- information identical to that of an ordinary Petri net : place and transition names,
- information similar to that of an ordinary Petri net, but with an enriched syntax : arc valuations, place markings,
- information that does not exist in the ordinary Petri net : place and transition domains, transition guards, transition priorities, delay or selection functions for the transitions.

This additional information is used for the definition of the semantics of an AMI net, which is briefly and informally described in the next section.

1.2. Behavior of an AMI net

The behavior of an AMI net is controlled by a set of rules that are identical to those used for general colored nets.

- A **domain** is associated with each place and transition of the model. The elements of these domains are called **colors**.
- When firing, a transition is binded by an element of its domain.
- Each token in a place is colored by an element of the place domain (several tokens may have the same color). The marking of a place is thus a **multiset** of colors - a set in which an element may occur several times.
- For a binded transition to be enabled, each input place of the transition must contain a sufficient (possibly null) number of tokens for every color of the place domain. These tokens will be taken from the place when the transition fires. Similarly, the firing will produce colored tokens in the output places of the transition. Like in Petri nets, the valuation attached to the arcs determines the number of tokens to be taken or produced. However, this valuation is now a **color function** that associates a multiset of colors of the place domain with each binding of the transition.
- Independently from the evaluation of the color functions, a transition may not be enabled if its binding does not satisfy some predicate. This predicate is called the **guard of the transition**.
- In the same way, a transition will not be enabled if a higher **priority** transition is enabled.

- Finally, as far as timed models are concerned - stochastic AMI nets - two kinds of transitions are considered : **timed transitions** with priority zero and **immediate transitions** with higher priorities. A random function, which determines the **firing delay**, is associated with a timed transition, and a **weight** is associated with an immediate transition. In a given marking, if several timed transitions are enabled, the one to fire is chosen according to a race policy (the transition that samples the least delay fires first). If several immediate transitions are enabled (they have the same priority), the one to fire is randomly chosen according to the probabilities that are derived from the respective weights of the transitions.

2. DECLARATION OF AN AMI NET

The description of an AMI net is made of two parts. The first part contains the declaration of the Petri net (places, transitions and arcs). This declaration is not included in the present document. The reader may refer to MACAO users' guide. We are only interested in the second part, i.e., the description of the high-level features of the model for which we give the syntactic construction rules.

Those high-level features have been listed in the former section. However, their description, i.e., the description of the domains, the markings, the guards, the priorities together with the color, delay or weight functions has not been presented.

These descriptions are done in different sections that are all optional :

- The **class declaration** section defines object classes. The objects are the elementary entities (sites, memories, etc…) that appear in the description of the model
- The **domain declaration** section defines the set of color domains associated with the different places and transitions of the net. A color associates one or several objects.
- The **variable declaration** section defines the name and the domain to which variables used for the valuation of the arcs belong.
- The **function declaration** section defines the name, the parameters, the expression, the domain and the codomain of functions that are used for the valuation of the arcs. Some functions that are often used are predefined in the analyzer.

In the rest of the document, keywords are written with bold characters.

2.1. Class Declaration

The keyword **CLASS** determines the beginning of the section in which the object classes that appear in the net are defined. All the classes must have different names : a class cannot be declared several times.

A class is defined by its name (identifier) and the set of objects it contains. This set of objects can be described either by an interval, or by an enumeration of the objects, possibly combined with an interval. **INTEGER** and **CHAR**, respectively the set of integers and the set of ASCII characters, are predefined in the analyzer.

Example

```
CLASS
     C1 IS INTEGER ;
     C2 IS [ a, b, c0..c9 ] ;
     C3 IS CHAR ;
     C4 IS [ 1..10 ] ;
```

When all the objects belonging to a class do not have the same potential behavior (some actions are imposssible for a subset of the objects of the class), the subsets of objects with the same behavior must be isolated in the class. These subsets are called static subclasses.

Example

```
CLASS
     fruits IS [ apple, pear, cherry, banana, pineapple ];
          european   STATIC(fruits) IS   [apple, pear, cherry]
          exotic     STATIC(fruits) IS   [banana, pineapple];
```

A class (except **INTEGER** and **CHAR**) can be partitionned (**SPLIT**) in one or several subsets of objects. Each subclass is characterized by its name and the number of objects it contains (this number can be omitted if the subclass contains only one element). The partition is not binded, i.e., the identity of the objects contained in each subclass is unknown. The subclass resulting from this partition are called dynamic subclasses.

This decomposition may be useful for instance if we want to declare that, in the initial state of the system, the ripe fruits, whatever their identitiy, can be eaten, whereas the green fruits still have to mature.

Example

```
CLASS
     fruits IS [ apple, pear, cherry, banana, pineapple ];
          european   STATIC(fruits) IS   [apple, pear, cherry]
                     SPLIT [ ripe(2), green(1) ] ;
          exotic     STATIC(fruits) IS   [banana, pineapple];
```

Such a declaration allows us to know that two european fruits are mature, but does not precise which ones.

It is difficult to count the elements in an interval. Hence, a class whose definition contains an interval can be split only if the identifiers of the bounds of the interval have the same alphabetic value.

It is possible to refer to an element in a class either directly by using its name *Classname.element*, or indirectly by giving its position in a subclass *Classname.part(n)*.

Example

```
fruits.pear
fruits.ripe(1)
```

fruits.pear is an unambiguous reference to the element pear of class fruits, whereas fruits.ripe(1) refers to the first element of the unbinded subclass ripe.

Several classes having the same definition can be declared at the same time. This facility will be useful when the further extensions of the language allow classes with the same definition to have different properties.

Example

```
CLASS
     C1, C2, C3 IS [ a, b, c ] ;
```

Remarks

- A class can contain any positive number of elements.
- The fact that a class is ordered or not is deduced from the color functions that appear in the net. In that case, the order of the elements is given by their enumeration order in the declaration.
- A static subclass can contain only elements that are contiguous with respect to the order relation.
- Objects belonging to different classes can have the same name.
- An identifier must begin by a letter or an underscore. An underscore alone is a correct identifier.
- The bounds of an interval are either numerical, or alphanumerical (identifiers). The left bound must be less than the right bound.
- For coherency reasons, a dynamic partition (**SPLIT**) can be done only inside static subclasses if there are some. However, the current version of the analyzer does not prevent the partitionning to be done directly at the class level.

2.2. Domain Declaration

The keyword **DOMAIN** determines the beginning of the section in which the domains of the places and the transitions that appear in the net are defined.

A domain can be simply a class, or can be built using set operators that apply on :

- classes defined in section **CLASS**
- domains already defined in section **DOMAIN**.

The possible operators are the union, the cartesian product and the powerset of a set (the set of all its subsets). A domain defined as a union is in fact a set of possible domains for a variable or an expression (see variable and function declaration).

Example

```
DOMAIN
       C1C2  IS <C1,C2>;
       D1    IS <C1,C2,C3>;
       PC1   IS PART(C1);
       C12   IS {PC1,C1C2};
```

Domain C1C2 is defined as the cartesian product of C1 and C2, PC1 is the powerset of C1 and C12 is the union of PC1 and C1C2.

The declaration of a domain using already defined classes or domains is done in the following way :

Example

```
DOMAIN
        D1 IS C1;
        D2 IS <C1,C2>;
        D3 IS <D2,C3>; {equivalent to D3 IS <C1,C2,C3>}
```

Like for the classes, several domains having the same definition can be declared at the same time.

Example

```
DOMAIN
        D1,D2,D3 IS <C1,C2>;
```

D1, D2, et D3 have the same definition.

Remarks

- There is no predefined domain.
- The place domains are defined explicitly, whereas for a transition the domain is derived from the examination of the functions around the transition (see § Function declaration).
- A domain containing a single element is called Null domain and no color is defined in it (a unique color gives no information on the identity).

2.3. Arc Valuation

The arcs of an AMI net are valuated by functions. The general expression of a function is a sum in which each term is the product of a guard by an elementary function.

The expression of an elementary function or a guard uses variables. Moreover, the analyzer includes predefined functions. We will hence examine successively

- the declaration of variables,
- the elementary functions,
- the guards,
- the arc valuation, i.e., the expression of complex functions.

2.3.1. Declaration of variables

The keyword **VAR** determines the beginning of the section in which variables are defined. It is not necessary to declare all the variables that appear in the expressions labelling the arcs of the net. The domain of an undeclared variable will be derived from the information on the place connected to the arc.

A domain is associated with each variable. This domain is either a domain defined in section **DOMAIN**, or built on domains defined in that section.

Example

```
VAR
  X12 IN <C1,C2>;
  X12bis IN D2;
  XPC1, YPC1 IN PC1;
```

Be careful ! Although it is possible to build its domain when declaring a variable, it is much wiser to use only domains that have already been declared in section **DOMAIN**. Actually, the consequences when using undeclared domains may be rather surprising. For instance, variables X12 and X12bis are not considered by the analyzer as belonging to the same domain.

Remarks

- A variable can be declared only once.
- A variable whose domain is a union of domains can take its value in any domain belonging to the union.
- An undeclared variable must belong to a unique domain and certainly not to a union of domains.

Like for classes and domains, several variables belonging to the same domain can be declared at the same time.

2.3.2. Declaration of elementary functions

The keyword **FUNCTION** determines the beginning of the section in which functions are defined. This section allows the designer to define new functions based on the functions predefined in the analyzer.

For AMI nets, the predefined functions are :

- numerical constants :

  1,2,3… (positive integers)

- class elements :

  C.c the element c in class C

  C.**ALL** all the elements in class C

  C.dyn(n) the n^{th} element of dynamic subclass dyn in class C

- elementary operations on the variables :

  X , X++n, X--n X, the n^{th} successor de X, the n^{th} predecessor of X. These operations can be applied only if the domain of X is a class.

  X#n the n^{th} component of X if X is a variable whose domain is a cartesian product.

Using the predefined functions, it is possible to define the elementary functions of AMI nets. These functions are :

- one of the foregoing expressions,
- the product of an integer by an expression,
- the sum or the difference of two expressions,
- a n-tuple in which each component is an expression,
- the union, the intersection or the set difference of two expressions,
- the operation **SET** (X) that changes X in {X}.

Generally speaking, the declaration of a function comprises a header, a body and a codomain. However, the codomain may not be specified.
The header includes the function name, together with its parameters and their domains. The body is an elementary expression as defined above. Finally, the codomain, if specified, is either a domain defined in section **DOMAIN**, or a domain built on domains defined in section **DOMAIN**.

Example

```
FUNCTION
        F(X:C1, Y:C2) TO D2 IS <C1.ALL,Y>;
        DIFFUSION(X:C1) IS C1.ALL;
        G(X:D1,Y:D2) IS <X,Y#2>;
```

Parameter Y is a variable belonging to domain D2 (see § 2.2), hence the structure of Y is <Y1,Y2>. The notation Y#2 denotes the element Y2 of variable Y.

The second declaration redefines diffusion in class C1. Although diffusion is a function without parameter, in the current version of the analyzer, a fictive parameter is required in the definition (this should be modified in future versions).

It may be interesting to define the functions that are frequently used, such as the following one that includes all the elements in a class, but one.

```
FUNCTION
      DIFF2(X:C1) IS C1.ALL-X;
```

Remarks

- All variables that appear in the body of a function must have been declared in the header of the function.
- When not specified, the codomain of a function is derived from its expression.
- Of course, the union, the intersection or the difference of two sets only apply to expressions of set type.
- Operation **SET** (X) can apply only to a class element or to a variable.

2.3.3. Guards

In an AMI net, guards can be associated either with transitions of with elementary color functions. The aim of a guard is always the same, i.e., to restrict the possible bindings of a transition (or the application of a color function) by adding constraints on the variables.

The guards are not defined in a particular section. They only appear associated with a transition or in the valuation expression of an arc.

The guards included in the definition of AMI nets can :

- demand that a variable belong to a specific static subclass or to a sub-domain of its domain,
- compare ($<, \leq, >, \geq, \neq$) two variables belonging to the same class, knowing that the order of the elements is the enumeration order in the class definition.

Guards can be combined using usual logical connectors AND, OR and NOT. It is also possible to add parentheses in their expression.

Remark

- C.**ALL** cannot be compared with another element by means of a relational operator.

2.3.4. Complex functions

The complex functions are the expressions used for arc valuation. They are built as a sum in which each term is the product of a guard by an elementary function. The grammar of valuation expressions is thus a super-set of the grammar used for the declaration of functions.

The evaluation of an expression is done term by term, and the result is the sum of the values obtained for each term. The evaluation of a term is done in the following way : if the guard is not true, then the term is null, else the elementary function is evaluated.

Valuation expressions can be either elementary expressions, or a conjuction of elementary expressions, each of them guarded. The conjunction operator is **`AND`**. A valuation expression cannot combine guarded and unguarded expressions.

Remark

- Empty guard [] whose value is always true can be used to combine a guarded function with an elementary function

Example

```
X + X++2
<X,Y++>
<X.ALL, C.a>
[X IN C1].X AND [X IN C2].C2.ALL-X
```

```
[X IN C1 OR Y IN C1].<C1.ALL-X,C1.ALL-Y> AND [NOT(X IN C1 OR
Y IN C1)].<X,Y>
[X IN C1 OR (X IN C2 AND Y=a)].<X,Y> AND [Y>a].<X,C1.a>
```

3. MACAO AND AMI NETS

This section describes how AMI nets can be designed using MACAO editor. We describe all different available nodes and the associated information. For each information, we indicate the default value if any. Otherwise, "" is assumed by MACAO editor.

3.1. Declaration node

This node contains the following information :

- declaration of classes, domains and variables using the syntax described in Sections 2.1, 2.2 and 2.3.1,
- author, version, project, title are information that the designer uses at his convenience.

3.2 Places

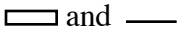
There are two different kinds of places : ordinary places and queues (FIFO places).

 These nodes contain the following information :

- name : a string value is expected,
- domain (default null) : null represents the absence of color domain. Otherwise, it must have been defined in the Declaration node,
- marking : an elementary function including only constants, or class elements (except C.dyn(n), see § 2.3.2)
- symbolic : not yet implemented
- component : used as an extra data by prototyping tools in order to define interfaces between submodels.

3.3. Transitions

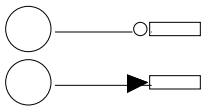
There are two different kinds of transitions : transitions with an exponentially distributed firing delay, and immediate transitions that fire in zero time. The information on timing and priorities is ignored when studying qualitative properties.

 These nodes contain the following information :

- name : a string value is expected,
- guard (default true) : the guard associated with the transition that must respect the syntax defined in Sections 2.3.3 and 2.3.4,

- priority : the default value for timed transitions is zero. The default value for immediate transitions is one,
- delay/weight : for a timed transition, gives the mean of the exponential distribution. For an immediate transition, the firing probability is given by the ratio of of the transition weight to the sum of the weights of all enabled transitions,
- component : not used for the while.

3.3. Arcs



These arcs contain the following information :

- value (default 1) : contains the valuation of the arc that must respect the syntax defined in Sections 2.3.2 and 2.3.4. The default value assumes that the place connected to the arc has a null domain.