A tool for symbolic manipulation of arc functions in Symmetric Net models

Lorenzo Capra
Università di Milano
Dipartimento di Informatica
Milano, Italy

Massimiliano De Pierro
Università di Torino
Dipartimento di Informatica
Torino, Italy

Giuliana Franceschinis
Univ. del Piemonte Orientale
Di.S.I.T.
Alessandria, Italy

ABSTRACT

The computation of structural properties of models expressed with the Symmetric Nets formalism (formerly Well-Formed Nets, a High Level Petri Net formalism), their structural reduction, or the efficient detection of transition instances enabled in a given state can benefit from the availability of a calculus for symbolic manipulation of arc functions. In previous works the theoretical basis of such calculus has been presented. In this paper a library implementing the calculus is described, and its use is demonstrated on a simple distributed system model.

1. INTRODUCTION

High Level Petri Nets (HLPN) formalisms have been proposed as extensions of the original Petri Net (PN) formalism to ease the task of modellers in designing complex systems. These mathematically sound graphical languages feature several different algorithms for studying interesting properties of systems. The analysis of HLPN models is usually performed through simulation or state space exploration techniques (e.g., model checking), but it can also work on their graph structure: structural analysis can be performed preliminarily to state space generation (e.g., to ensure boundedness), as a quick check for (un)desired behaviours (e.g., livelocks), as a model reduction technique [4] or as a support to specification of model parameters in stochastic HLPN (detecting potentially conflicting transitions [2] is a prerequisite for a correct parameters setting). Finally, structural information can significantly improve the efficiency of both simulation and state space generation.

The success of these formalisms is also motivated by the availability of software tools supporting design and analysis of models[1]. In this paper a tool for structural analysis of Symmetric Nets (SN) [1] is described. The tool builds on a language $\mathcal{L}$ extending the SN arc expressions, by which various kinds of structural properties can be expressed in a compact and parametric form.

The tool implements a rewriting system able to transform the complex expressions representing structural formulae (involving arc functions, transition guards, as well as functional operators such as intersection, union, transpose, and composition), into a normalized form in $\mathcal{L}$. Since $\mathcal{L}$ has a syntax very similar to that of SN arc expressions, its interpretation does not require too much effort. The task of normalizing an expression is in general long and error prone, so that its complete automation is a prerequisite for applicability of a structural calculus to SNs.

2. AN OVERVIEW OF SN

Figure 1 shows an example of a SN modeling a simple distributed system. The SN structure is a bipartite graph whose nodes are *places* (circles) and *transitions* (white and black bars). Places are state variables, characterized by a *color domain* defining the variables' type and expressed as a Cartesian product of *basic color classes* (pairwise disjoint finite sets, denoted with capital letters A, B, ..., Z, which may be partitioned into two or more *static subclasses*, and may be circularly ordered). Each place can contain a multiset of tuples from its color domain: this is called its *marking*. The SN of the example represents the behavior of k *sites*, whose identities correspond to the colors in class $A = \{\mathtt{s}_1, \ldots, \mathtt{s}_k\}$. The marking of place `Sites` indicates the active sites: initially (marking $\mathbf{m}_0$) all sites are in this state. Each *site* sends some data, embedding it in a sequence of packets. Packets are represented by color class M which is partitioned into $M_1 \cup M_2$: $M_1 = \{\mathtt{ack}\}$ represents an acknowledge message while $M_2 = \{\mathtt{d}_1, \ldots, \mathtt{d}_h\}$ represents data packets. Each data packet is associated with a header containing the sender and destination identifiers, and sent through the network. The subnet between places `T-buffer` and `R-buffer` models the network and contains messages stored in the transmit/receive buffers, represented by tuples $\langle \mathtt{s}_i, \mathtt{s}_j, \mathtt{d}_k \rangle$ whose first, second, and third element identify the sender, ($\mathtt{s}_i \in A$), the destination ($\mathtt{s}_j \in A$), and the data packet ($\mathtt{d}_k \in M$), respectively.

Also transitions have a color domain, as they describe parametric events; the parameters are variables denoted with small letters with a subscript, implicitly defining the variable's type: the color class denoted by the corresponding capital letter; subscripts are thus used to distinguish parameters of same type associated with the same transition. Transitions can have guards, expressed in terms of predicates on the transition's variables.

[1] Petri Nets Tool Database http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html. Petri Nets Model Checking Contest http://mcc.lip6.fr

ValueTools'13, December 10 – 12 2013, Turin, Italy
.

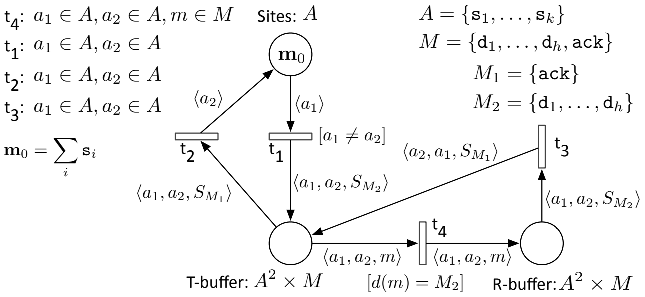
t_4: $a_1 \in A, a_2 \in A, m \in M$
t_1: $a_1 \in A, a_2 \in A$
t_2: $a_1 \in A, a_2 \in A$
t_3: $a_1 \in A, a_2 \in A$
$\mathbf{m}_0 = \sum_i \mathtt{s}_i$
Sites: A
$\mathbf{m}_0$
$A = \{\mathtt{s}_1, \ldots, \mathtt{s}_k\}$
$M = \{\mathtt{d}_1, \ldots, \mathtt{d}_h, \mathtt{ack}\}$
$M_1 = \{\mathtt{ack}\}$
$M_2 = \{\mathtt{d}_1, \ldots, \mathtt{d}_h\}$
$\langle a_2 \rangle$
$\langle a_1 \rangle$
t_2
t_1
$[a_1 \neq a_2]$
$\langle a_2, a_1, S_{M_1} \rangle$
t_3
$\langle a_1, a_2, S_{M_1} \rangle$
$\langle a_1, a_2, S_{M_2} \rangle$
$\langle a_1, a_2, S_{M_2} \rangle$
t_4
$\langle a_1, a_2, m \rangle$
$\langle a_1, a_2, m \rangle$
T-buffer: $A^2 \times M$
$[d(m) = M_2]$
R-buffer: $A^2 \times M$

Figure 1: An example of communication system.

Transitions $\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_3$ have color domain A^2, their variables are shown in Figure 1. Transition $\mathtt{t}_1$ represents the sending of data packets by site a_1 to site a_2, with $a_1 \neq a_2$. Transition $\mathtt{t}_4$ has color domain $A^2 \times M$ and represents the transmission of each packet queued at the network transmission buffer. Transition $\mathtt{t}_3$ represents the receiver process active on each site, it collects all the data packets sent to it; once all packets from a given sender have arrived, it sends back to the sender an acknowledge message, represented by a tuple with parametric sender and destination $a_2 \in A$, $a_1 \in A$ and content $\mathtt{ack} \in M_1$. Transition $\mathtt{t}_2$ ends the transmission process: when the site that started data transmission receives the acknowledge packet, it moves from the waiting state back to the active one, where it can decide to send other data packets out.

The model evolution in time can be simulated by starting from an initial marking and firing one of the enabled *transition instances*. An instance of transition t is an element of t's color domain ($cd(t)$)) and corresponds to an assignment of colors to t's parameters (binding). A binding is valid only if it satisfies the transition guard. The arcs connecting transitions to their input, output and inhibitor places are annotated with functions (denoted by $W^-(p,t)$, $W^+(p,t)$ and $W^h(p,t)$, respectively) $cd(t) \to Bag(cd(p))$. Input and inhibitor functions express the transition enabling conditions, while the difference $\mathbf{C}(p,t) = W^+(p,t)(b) - W^-(p,t)(b)$ defines the effect on p of firing the (enabled) instance b of t.

The particular syntax of SN arc functions and guards highlights the structural and behavioral symmetry of the modeled system: this can be exploited for analysis purposes.

Definition 1 (Guards syntax). *Guards are boolean expressions whose terms are* basic predicates. *The set of basic predicates is:* $[var1 = var2]$, *true when* $var1$ *and* $var2$ *are bound to the same color;* $[var1 = !var2]$, *true when* $var1$ *is bound to the successor of the color assigned to* $var2$*;* $[d(var1) = S_{subclass\_id}]$, *true when the color assigned to* $var1$ *belongs to static subclass subclass_id), and* $[d(var1) = d(var2)]$, *true when the colors assigned to* $var1$ *and* $var2$ *belong to the same static subclass.*

Definition 2 (Arc functions syntax). *A SN function W labeling an (input, output or inhibitor) arc connecting transition t and place p, is a mapping* $W(p,t) : cd(t) \to Bag\big(cd(p)\big)$ *whose form is:*

$$W(p,t) = \sum_i \lambda_i . T_i[p_i], \quad \lambda_i \in \mathbb{N}^+ \tag{1}$$

where the sum is a multiset sum and λ_i *are scalars,* $T_i = \langle f_1, \ldots, f_n \rangle$ *are tuples of class functions, and* p_i *is a guard. Class functions syntax (referring to class C) is:*

$$f_i = \sum_{k=1}^{m} \alpha_k . c_k + \sum_{q=1}^{||C||} \beta_q . S_{C_q} + \sum_{k=1}^{m} \gamma_k . !c_k; \; \alpha_k, \beta_k, \gamma_k \in \mathbb{Z} \tag{2}$$

In (2) scalars $\alpha_k, \beta_k, \gamma_k$ *must be such that no negative coefficient result by evaluating* f_i *for any color satisfying* p_i*.*

Summarizing, an arc function is a weighted sum of possibly guarded tuples (T_i) of class functions (f_k). Class functions are linear combinations of: projection (c_k), successor ($!c_k$, defined only for ordered classes), synchronization/diffusion (S_{class_id}) function. An arc function is evaluated on a binding $b \in cd(t)$, and $\langle f_1 \ldots f_n \rangle(b) = \bigotimes_{k:1\ldots n} f_k(b)$ (where $\bigotimes$ denotes the Cartesian product). The projection evaluates to the color assigned to the corresponding variable, the successor evaluates to the successor of that color, the diffusion/synchronization function evaluates to the whole set of elements in *class_id*. If class C is partitioned into *static subclasses* C_i, $1 \leq i \leq ||C||$, it is possible to use the diffusion/synchronization function on a static subclass (S_{C_i}).

A guarded tuple is evaluated as follows: if for a given binding the guard is false it evaluates to the empty (multi)set, otherwise its value corresponds to its standard evaluation.

An example of valid binding for $\mathtt{t}_1$ is $b : (a_1 = \mathtt{s}_i, a_2 = \mathtt{s}_j)$: it satisfies the guard $[a_1 \neq a_2]$. When enabled it may fire, consuming a token $\langle a_1 \rangle(b) = \langle \mathtt{s}_i \rangle$ from `Sites` and producing $\langle a_1, a_2, S_{M_2} \rangle(b) = \sum_{k=1\ldots h} \langle \mathtt{s}_i, \mathtt{s}_j, \mathtt{d}_k \rangle$ in `T-buffer`.

3. THE LANGUAGE

In this section the syntax of the language used to express the SN structural relations is introduced, together with the set of operators that the symbolic calculus handled by the library can deal with. The expressions of $\mathcal{L}$ have a syntax which resembles the arc function syntax defined in equations (1) and (2), but there are some additional constraints on the functions used as elementary building blocks, however the expressive power is actually extended.

Let $\Sigma = \{A, B, \ldots, Z\}$ be the set of basic color classes. A class function in the new language is any function with domain $\mathcal{D}$ (expressed as Cartesian product of basic color classes in Σ) and expressed as sum of *intersections* of the following elementary symbols:

$$\{var, S_{subclass}, S_{class}, S - var, !^n var, S - !^n var\} \tag{3}$$

Some class function examples follow: $(S - a_1) \cap (S - a_2)$; $a_2 \cap (S - a_3)$; $m_3 \cap !^4 m_1$; $d_1 + (S - !d_3)$.

Definition 3 (Language). *Let* $\Sigma = \{A, B, \ldots, Z\}$ *be the set of (finite and disjoint) basic color classes, and let* $\mathcal{D}$ *be any color domain built as Cartesian product of classes in* Σ*,* ($\mathcal{D} = A^{e_A} \times B^{e_B} \times \ldots \times Z^{e_Z}, e_* \in \mathbb{N}$)*. Let* T_i *be functions on* $\mathcal{D}$ *with values in* $\mathcal{D}'$ *and* $[g_i']$ *and* $[g_i]$ *standard predicates respectively on* $\mathcal{D}'$ *and* $\mathcal{D}$ *(as defined in the SN formalism).*

The set of expressions:

$$\mathcal{L} = \Big\{ F : F = \sum_i \lambda_i . [g_i'] T_i [g_i], \quad \lambda_i \in \mathbb{N}^+ \Big\}$$

is the language used to express SN structural relations (where the $T_i = \langle f_1, \ldots, f_l \rangle$ *are composed of class functions* f_i*: intersections of elementary functions listed in (3)).*

The main difference with respect to SN arc functions is the use of intersection in class functions, and the presence of two predicates associated with each tuple in the sum: $[g_i]$ is called *guard* and is already present in Eq. (1) while $[g'_i]$ is called *filter*, a new feature of the language that allows the elements satisfying predicate g'_i to be selected from the result of the application of the guarded tuple $T_i[g_i]$.

It is possible to show that the SN arc functions $W^-(p,t)$, $W^+(p,t)$, $W^h(p,t)$ can be reformulated as elements of $\mathcal{L}$. On $\mathcal{L}$ the following functional operators are defined:

Operator	Semantics	Operator	Semantics
$\overline{F}$	Support	$F \cap F'$	Intersection
$F - F'$	Difference	$F + F'$	Sum
F^t	Transpose	$\overline{F} \circ \overline{F'}$	Composition

All operators in the above table, except composition, apply to functions that map to multisets. The composition operator instead, in its current definition and implementation, applies only to functions that map to sets (this is why in the table the support of composition operands is used). Hence the composition actually works on a subset of $\mathcal{L}$.

In the sequel the term *expression* will be used to indicate formulae that contain language functions and operators from the table above. The symbolic calculus implemented by the library is able to *solve* all the considered operators: appropriate rewriting rules have been defined that simplify the expressions containing any operator until an element of $\mathcal{L}$ is obtained. Hence the language is closed w.r.t. the above operators. Each rewriting rule is based on the algebraic properties of functions appearing as operands.

A detailed description of these rules can be found in [2], where the difference, intersection, transpose operators rewriting rules have been first introduced.

4. THE TOOL FRONTEND

The net in Fig. 1 will be used to show the application of the calculus implemented by the tool. In particular the examples will concern the verification of colored invariance properties and the computation of structural conflicts. The interaction with the library is performed through a Command Line Interface (CLI).

The verification of structural invariance properties of the model is based on the *incidence matrix*: the one of the model in Fig.1 is shown in Tab. 1. We will verify that the following expression corresponds to a structural invariant of the distributed system model:

$$\langle a_1, S_{M_2}\rangle.\texttt{Sites}+\langle a_1, m\rangle.\texttt{R-buffer}+\langle a_1, m\rangle[d(m)=M_2].\texttt{T-buffer}+\langle a_2, S_{M_2}\rangle[d(m)=M_1].\texttt{T-buffer} = K$$

where K is a constant multiset on $A \times M$ (color domain of the invariant), which depends on the initial marking $\mathbf{m}_0$. It is possible to express the invariant as a P-indexed array:

$$\begin{array}{cccc} & \texttt{Sites} & \texttt{T-buffer} & \texttt{R-buffer} \\ \mathbf{i}: & \langle a_1, S_{M_2}\rangle & \langle a_1, m\rangle[d(m)=M_2]+ & \langle a_1, m\rangle \\ & & +\,\langle a_2, S_{M_2}\rangle[d(m)=M_1] & \end{array}$$

We need to verify the following matrix equation: $\mathbf{i} \circ \mathbf{C} = \mathbf{0}$.

The second example shows how the calculus may be applied to identify potential conflicts between different instances of a given transition. Let us consider transition $\texttt{t}_1$, the following computation is performed through the library:

$$\begin{aligned} SC(\texttt{t}_1, \texttt{t}_1) &= \overline{W^+(\texttt{t}_1, p)^t \circ W^-(\texttt{t}_1, p)} - id = \\ &= \langle a_1\rangle[a_1 \neq a_2]^t \circ \langle a_1\rangle[a_1 \neq a_2] - id \end{aligned}$$

where id is the identity function in $A \times A$, i.e. $\langle a_1, a_2\rangle$. $SC(\texttt{t}_1, \texttt{t}_1)$ is a function that states which instances of $\texttt{t}_1$ may disable a given instance of the same transition.

The CLI implements a command-line interpreter that accepts in input any valid expression of the symbolic calculus and, using the library to apply the appropriate rewriting rules, returns a simplified expression of the language. It has been designed as a utility to test the library functionality, rather than with the aim of providing a user friendly environment, despite this, it does not restrict the possibility to experiment any practical case that may arise from the structural analysis of SN models. Let us introduce the syntax and grammar of the strings accepted by the CLI through examples. The following table lists all the operators that may appear in any expression to be simplified.

CLI	Operator	CLI	Predicate Operator
`*`	Intersection	`!=`	Unequality
`.`	Composition	`=`	Equality
`'`	Transpose	`in`	Memebership
`-`, `+`	Difference, Sum	`and`,`or`	Boolean AND, OR

The set of color classes is defined by default: there are 26 color classes identified by the (capital) letters of the alphabet A-Z. By default the classes are not ordered, are not partitioned into static subclasses and have a *parametric* size n where[2] $n \geq 2$. Commands are provided to modify the default definition of a class, according to the modeling requirements. For instance, referring to the distributed system model introduced in the previous section, class M can be defined using the following syntax: `set M := {1,[2,n]}`, meaning that M comprises two static subclasses: the first (M_1) has size 1, while the second (M_2) has parametric size n, with $n \geq 2$. In general, the size of each static subclass can be either fixed, or parametric: in the latter case it is specified using an interval notation indicating the lower and upper bound for the size of the subclass; the upper bound can be fixed or parametric, and in the latter case it is denoted by `n`. The library currently requires that at most one static subclass has parametric size (an extension relaxing this constraint is under study). An ordered class is instead declared through the command `set N ordered`; it cannot be partitioned into static subclasses.

The CLI maintains a table of symbols associated with either *functions* or *color domains*. This is useful for an easier input of complex formulae of the calculus, and for reusing recurring expressions.

An example of association of a color domain to a symbol is `D:=@A^2,M`. Color domain definitions always start with symbol `@`, the classes in the color domain must appear in alphabetical order (a requirement imposed by the library), moreover the class repetitions are expressed using the `^` symbol followed by the number of repetitions.

The association of language expressions to symbols is performed as follows[3]:

```
i1 := @A      <a_1,S_M{2}>
i2 := @A^2,M <a_1,m_1>[m_1 in M{2}] + <a_2,S_M
    {2}>[m_1 in M{1}]
i3 := @A^2,M <a_1,m_1>
```

Observe that each expression must be preceded by the domain of the corresponding function, e.g. function `i2` has domain $A^2 \times M$ (the library performs a consistency check on the domains of the class functions that appear in the expression). The expressions associated with symbols `i1`, `i2` and `i3` correspond to the three elements of vector $\mathbf{i}$, and will turn useful to verify the system of equations $\mathbf{i} \circ \mathbf{C} = \mathbf{0}$.

[2] The library restriction on the minimal size of color classes is motivated by the observation that cardinality one color classes can be removed without producing any change in the model behavior.

[3] Character `_` is used to introduce a subscript, while `^` introduces superscripts.

Table 1: Incidence matrix

$$\mathbf{C} = \begin{array}{ccccl} \mathtt{t}_1 & \mathtt{t}_2 & \mathtt{t}_3 & \mathtt{t}_4 & \\ -\langle a_1\rangle & \langle a_2\rangle & 0 & 0 & \mathtt{Sites} \\ \langle a_1, a_2, S_{M_2}\rangle[a_1 \neq a_2] & -\langle a_1, a_2, S_{M_1}\rangle & \langle a_2, a_1, S_{M_1}\rangle & -\langle a_1, a_2, m\rangle[d(m) = M_2] & \mathtt{T\text{-}buffer} \\ 0 & 0 & -\langle a_1, a_2, S_{M_2}\rangle & \langle a_1, a_2, m\rangle[d(m) = M_2] & \mathtt{R\text{-}buffer} \end{array}$$

Observe that static subclasses are denoted by a numeric id within curly brackets following the class id, e.g. `M{2}` denotes static subclass M_2 in M. Concerning predicates (expressing filters and guards), the only difference w.r.t. the standard SN notation is the use of keyword `in` to denote the membership relation: e.g. SN predicate $[d(m_1) = M_2]$ becomes `[m_1 in M{2}]`.

To perform the calculus (simplification of the expressions appearing in the four equations, one for each column of $\mathbf{C}$) the CLI uses the built-in procedure `s()`: let `D:=@A^2,M`

```
s(D i2.<a_1,a_2,S_M{2}>[a_1!=a_2]-i1.<a_1>)
s(D i1.<a_2> - i2.<a_1,a_2,S_M{1}>)
s(@A^2 i2.<a_2,a_1,S_M{1}>-i3.<a_1,a_2,S_M{2}>)
s(D i3.<a_1,a_2,m_1>[m_1 in M{2}]-i2.<a_1,a_2,
    m_1>[m_1 in M{2}])
```

`s()` calls the library procedure that performs the *normalization* of an expression by using the algebraic rules of the *calculus*. It accepts in input any valid calculus expression (preceded by its color domain). Dot symbol `.` represents the composition operator. The rest of the syntax as well as the precedence rules of operators are the same as in classical algebra.All four expressions above are simplified and the composition operator removed, leading to the expected result: in all cases $< 0, 0 >$, a constant "null" function, confirming the fact that **i** denotes an invariant induced by the model structure, valid for any initial marking.

Let us show another example of computation of potential conflicts among instances of a given transition $\mathtt{t}_1$ (called auto-conflicts): $SC_{\mathtt{Sites}}(\mathtt{t}_1, \mathtt{t}_1)$. The following command sequence leads to the computation of such structural relation: the example shows the possibility to break an expression into simpler ones through the definition of symbols:

```
f  := @A^2 <a_1>[a_1!=a_2]
id := @A^2 <a_1,a_2>
s(f'.f - id)
```

where `f'` denotes the transpose of function `f` (i.e. `'` denotes the transpose operator). The result returned by the CLI is:

```
<0,0>                              : |A|=2
<a_1,S-a_1 * S-a_2> [a_1 != a_2]   : 3<=|A|<=n
```

Observe that in this case the result provides two different expressions, depending on the cardinality of parametric class A. The intersection operator (`*`) is used to represent certain legal SN functions: specifically the following equivalence holds: $(S{-}a_1 \cap S{-}a_2)[a_1 \neq a_2] \equiv (S{-}a_1{-}a_2)[a_1{\neq}a_2]$.

In the current version of the tool the function operands of all operators are interpreted as sets: work is in progress to allow multiset operands (except for composition, since this extension still requires some theoretical work).

5. LIBRARY ARCHITECTURE

The `Java` core library for the calculus, which interfaces to the CLI, implements a (parametric) symbolic rewriting system: a collection of rules (equations) which are used to rewrite (evaluate) expressions in a symbolic fashion, until no more rules apply, in which case the resulting term is in "normal form". For implementation convenience, the normal form of expressions manipulated by the library is a particular sum of pairwise disjoint terms belonging to $\mathcal{L}$. The Interpreter, one of the original Design Patterns for object oriented software, provides an elegant and natural way of implementing a rewriting system. Its twofold intent is, given a language, defining a representation for its grammar along with an interpretation engine that uses that representation to process sentences. A concrete class is used to represent each grammar's rule/symbol. A rule is either a composite object (a rule that references to other rules) or a terminal (a leaf node in a logical tree structure). The root of the hierarchy is an abstract type declaring a method `simplify()` that must be implemented by each concrete class. Reducing a term to a normal form relies on the recursive traversal of the underlying Composite structure.
In order to face design complexity, and to enable modular debugging/testing, three separated hierarchies of cooperating objects were defined, each formed by a few dozens classes: their root types are `ClassFunction`, `Guard`, and `TupleFunction`. This choice reflects the structure of SN's expressions.

Each hierarchy matches in fact an abstract "Boolean algebra", hence rewriting any (sub-)expression involves using logical equivalences such as the double complement elimination, De Morgan, associativity, commutativity, idempotence, and so on. An implementation based on the original Interpreter would result in a hard to maintain code, due to pollution of generic and domain-specific aspects. For that the library builds on a recent domain-parametric extension of Interpreter [3] that significantly helps design rewriting-based systems, by decoupling "generic" rewriting rules from type-specific ones. This approach heavily relies on generic (abstract) types and methods, and exploits other original patterns like Template Method, Factory Method, Command and Singleton. A detailed description can be found in [3].

6. CONCLUSION AND FUTURE WORK

The current version of the library and its CLI can be downloaded from http://www.di.unito.it/∼depierro/vt13. The library is being extended to support multiset operands for all operators. A Graphical User Interface will be developed, to allow the specification of the structural properties of interest directly on a graphical representation of the model.

7. REFERENCES

[1] *Standard published: ISO/IEC 15909-2:2011 Systems and software engineering - High-level Petri nets - Part 2: Transfer format.* See also http://pnml.lip6.fr.

[2] L. Capra, M. Pierro, and G. Franceschinis. A high level language for structural relations in Well-formed Nets. In *Applications and Theory of Petri Nets 2005*, volume 3536 of *LNCS*, pages 168–187. Springer, 2005.

[3] L. Capra and V. Stile. An extension of the interpreter pattern to define domain-parametric rewriting systems. In *Proc. 15th Int. SYNASC'13.* IEEE C.S.Press, 2013.

[4] S. Evangelista, S. Haddad, and J.-F. Pradat-Peyre. Syntactical colored Petri nets reductions. In *Automated Technology for Verification and Analysis*, volume 3707 of *LNCS*, pages 202–216. Springer, 2005.