Technical Report

Simple Colored Petri Nets

Antonio Camurri and Alessandro Coglio

Dept. of Informatics, Systems, and Telecommunications (DIST)
Faculty of Engineering – University of Genoa

Viale Causa 13
16145 Genova, Italy

E-mail: {music, tokamak}@dist.unige.it

November 1997

Simple Colored Petri Nets

Antonio Camurri and Alessandro Coglio

Abstract

Petri Nets (P-nets) are widely used in many fields. Because of their simple working, many formal analysis and synthesis methods are applicable to them, and furthermore P-net-based software tools can be implemented with relative ease. However, P-nets are inconvenient to use in very large and complex applications. Colored Petri Nets (CP-nets) provide much more convenience and compactness than P-nets. The price is that greater efforts are required to implement CP-net-based software tools, and that most formal analysis and synthesis methods are not applicable to them.

We propose *Simple Colored Petri Nets* (*SCP-nets*) as a promising trade-off between the simplicity of P-nets and the convenience of CP-nets. SCP-nets are similar to CP-nets, but user-defined data types must be enumerative. Despite this restriction, SCP-nets provide much more convenience than P-nets. Since operations upon enumerative types are most naturally defined by listing argument-result pairs, SCP-net-based software tools can be implemented much more easily than CP-net-based ones, because no general programming language is needed to define types and operations. Moreover, SCP-nets seem to be more amenable to formal analysis and synthesis methods than CP-nets, as we are investigating through some ongoing research. Another valuable feature of SCP-nets is that, unlike CP-nets, their types can be organized in hierarchies.

1. Introduction

Petri Nets (P-nets) [3, 2] are widely used as a modeling and specification tool in a variety of fields. The main reason for their success is that they can conveniently express pre- and post-conditions, distributedness, concurrency, synchronization, sharing, flows, etc., visually conveying such concepts through graphical representation. Furthermore, there exists a host of methods to perform formal analysis of various properties of P-nets (e.g. liveness, boundedness), which correspond to properties of the entities modeled or specified. There also exist methods to formally synthesize P-nets with certain properties (e.g. deadlock freeness) guaranteed by construction [4]. In addition, the relatively simple working of P-nets allows software tools based on P-nets (e.g. editors, executors, analyzers) to be implemented with relative ease.

However, as it is well-known, the size of P-nets tends to become very large as the complexity of the entities modeled or specified grows. Too large P-nets are inconvenient to manipulate, mainly because graphical cluttering hinders the conveying of visual information. Moreover, most methods of formal analysis cannot be applied to very large P-nets owing to computational complexity.

Colored Petri Nets (CP-nets) [1] overcome the size problem, as they allow even complex entities to be modeled and specified with great compactness and convenience. The other side of the coin is that most methods of formal analysis (and synthesis as well) for P-nets do not generalize to CP-nets. Moreover, implementing software tools based on CP-nets requires much greater efforts, owing to their relatively complex working.

In this report we propose *Simple Colored Petri Nets* (*SCP-nets*) as a promising trade-off between P-nets and CP-nets. SCP-nets are similar to CP-nets, with tokens being data items of data types, and with declarative expressions and guards on arcs and transitions defining the ways in which transitions can fire. However, user-defined types must be *enumerative* (in the sense of programming languages such as C or Pascal), i.e.

their items must be explicitly enumerated one by one. Despite this restriction, SCP-nets still provide much more convenience than P-nets, as an SCP-net can be "exponentially smaller" than an equivalent P-net. Furthermore, software tools based on SCP-nets can be implemented with much more ease than those based on CP-nets, because operations upon enumerative types are most naturally defined by enumerating argument-result pairs. Hence, unlike CP-nets, no general programming language is needed to define types and operations for SCP-nets. Moreover, SCP-nets seem to be more amenable to formal analysis and synthesis than CP-nets, as we are investigating through some ongoing research.

In Section 2 we formally present SCP-nets, and in Section 3 we draw some conclusions and outline future work. The mathematical concepts and notations used in Section 2 are explained in the Appendix.

2. Formal Presentation

As we present formal definitions in the following subsections, we instantiate them to the very simple SCP-net depicted in Figure 1, which, despite its simplicity, exhibits the key characteristics of SCP-nets.

2.1 Topology

Like P-nets and CP-nets, SCP-nets have places, transitions, and arcs connecting them, as formalized by the concept of topology.

Definition 1. A *topology* is a quadruple

$$\boldsymbol{T} = \langle P, T, A, \chi \rangle$$

where:
(1) P is a finite set whose elements are called *places*;
(2) T is a finite set whose elements are called *transitions*;
(3) A is a finite set whose elements are called *arcs*;
(4) $P \cap T = \varnothing$;
(5) $\chi \in [\, A \to (P \times T) \cup (T \times P) \,]$ and is called *connection*.
For each $a \in A$, let $p \in P$ and $t \in T$ such that $\chi(a) = \langle p, t \rangle$ or $\chi(a) = \langle t, p \rangle$; we define:
(1) $\chi_P(a) = p$;
(2) $\chi_T(a) = t$.
For each $x \in P \cup T$, we define the sets $In(x)$ and $Out(x)$ of the *incoming arcs* and *outgoing arcs* of x as follows:
(1) $In(x) = \{\, a \in A \mid \exists\, y \in P \cup T : \chi(a) = \langle y, x \rangle \,\}$;
(2) $Out(x) = \{\, a \in A \mid \exists\, y \in P \cup T : \chi(a) = \langle x, y \rangle \,\}$.
For each $p \in P$ and $t \in T$, we say that:
(1) p is an *input place* of t and t is an *output transition* of p iff $In(t) \cap Out(p) \neq \varnothing$;
(2) p is an *output place* of t and t is an *input transition* of p iff $In(p) \cap Out(t) \neq \varnothing$.

The connection χ associates a pair $\langle x, y \rangle$, where either x is a place and y a transition or vice versa, to each arc, meaning that the arc goes from x to y. In order to avoid ambiguities, P and T must be disjoint. The functions χ_P and χ_T respectively map each arc to the place and transition connected by the arc. A place and a transition may be connected in the same direction by two or more distinct arcs.

In Figure 1 the topology consists of three places (P0, P1, and P2) represented as circles, two transitions (T1 and T2) represented as rectangles, and four arcs (A1, A2, A3, and A4) represented as arrows, with the connection defined as graphically shown.

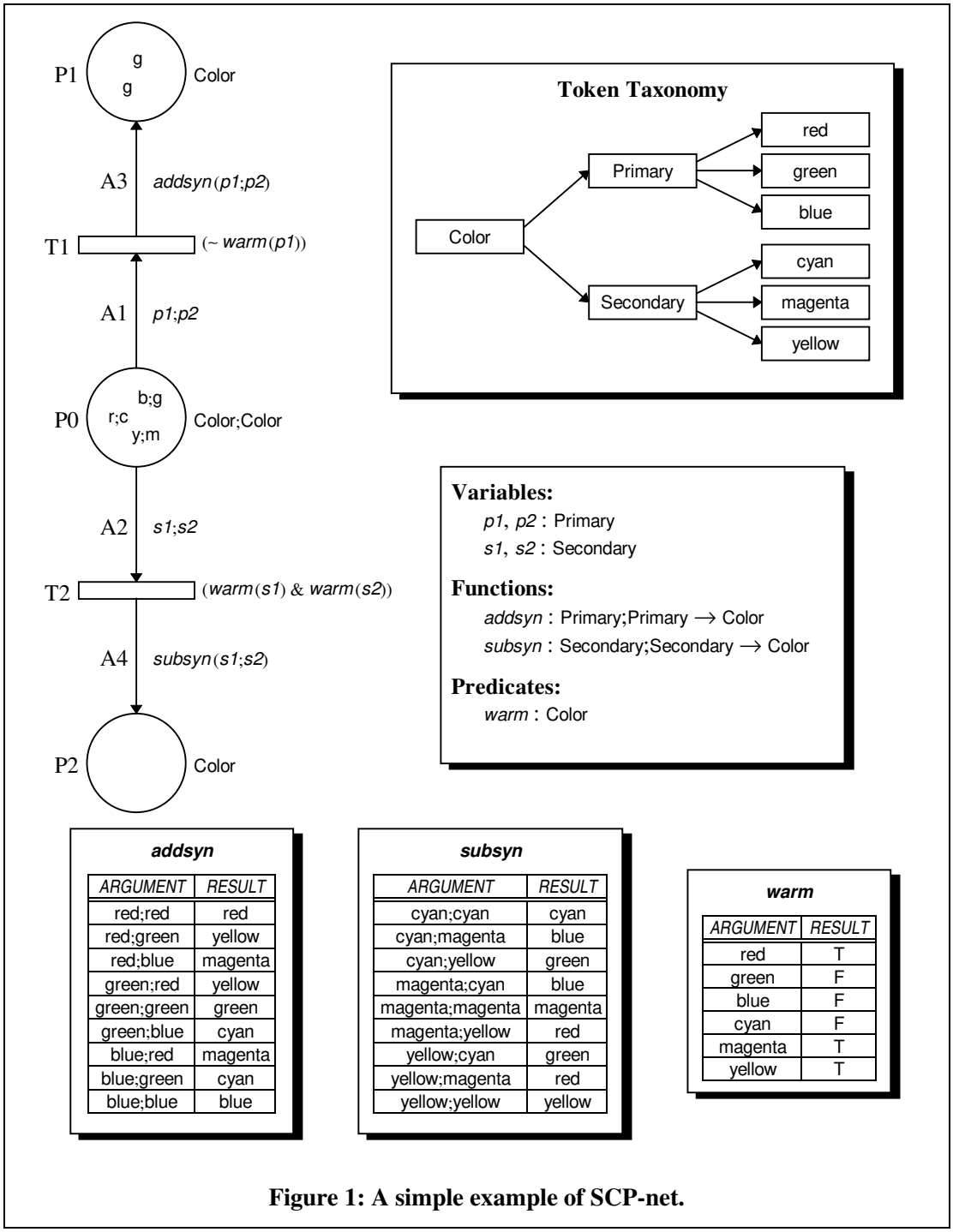

Figure 1: A simple example of SCP-net.

2.2 Token Taxonomy

In an SCP-net there are user-defined enumerative types, as mentioned in Section 1. A token is a finite non-empty concatenation of enumerative items. The components of the concatenation can model or specify discrete attributes of the entity modeled or specified by the whole token. For instance, in an automated manufacturing scenario there might be tokens representing manufacturing parts whose components represent geometrical shapes, constituent materials, chromatic colors, etc.

Instead of just having a flat collection of types as in CP-nets, the types of an SCP-net can be organized in hierarchies. A type can have super-types and sub-types,

whose tokens are respectively supersets and subsets of the tokens of the type. For instance, a type for manufacturing parts might specialize into sub-types for wooden parts, metallic parts, etc., and these might further specialize according to the kinds of wood, metal, etc. The type hierarchy is graph-shaped in general, and not just tree-shaped. For instance, the same type for manufacturing parts might specialize into a tree of sub-types according to shapes or colors, thus in fact having partially overlapping trees of types.

In an SCP-net, user-defined enumerative types and their items are represented as symbols (i.e. identifiers). Membership of items to types and hierarchies of types are specified by directed edges from type symbols to item symbols or to other type symbols: an edge from a type to an item specifies that the item has that type; an edge from a type to another type specifies that the first is a super-type of the second (and symmetrically the second is a sub-type of the first), implying that all the items of the second type are also items of the first. These symbols and edges thus make up a directed graph. Tokens are represented as strings of one or more concatenated item symbols, and accordingly types of tokens are represented as strings of concatenated type symbols. Membership and hierarchy lift from user-defined types and items to type and token strings, in the natural way (i.e. "component-wise"). All these ideas are formalized by the concept of token taxonomy.

Definition 2. A *token taxonomy* $\boldsymbol{X}$ is a finite non-empty DAG of symbols, such that no symbol is isolated in the DAG.
The set Y_B of *base types* is

$$Y_B = \mathrm{NTerm}(\boldsymbol{X}).$$

For each $y, y' \in Y_B$, we say that y is a *super-type* of y' and that y' is a *sub-type* of y iff

$$y \rightarrow_X y'.$$

The set Y of *types* is the smallest set such that

$$Y = Y_B \cup \{\ y;y'\ \text{string} \mid y, y' \in Y\ \}.$$

For each $y, y' \in Y$, we say that y is a *super-type* of y' and that y' is a *sub-type* of y iff for some $y_1, \ldots, y_n, y_1', \ldots, y_n' \in Y_B$ we have:
(1) $y = y_1;\ldots;y_n \ \wedge\ y' = y_1';\ldots;y_n'$;
(2) $\forall\ i \in \{1,\ldots,n\} : (\ y_i = y_i' \vee y_i$ is a super-type of $y_i'\ )$;
(3) $\exists\ i \in \{1,\ldots,n\} : y_i \neq y_i'$.
The family $\{K_y\}_{y \in Y}$ of *tokens* is the smallest family such that:
(1) $\forall\ y \in Y_B : K_y = \{\ k \mid k \in \mathrm{Term}(\boldsymbol{X}) \wedge y \rightarrow_X k\ \}$;
(2) $\forall\ y, y' \in Y : K_{y;y'} = \{\ k;k'\ \text{string} \mid k \in K_y \wedge k' \in K_{y'}\ \}$.
The set K of all tokens is thus

$$K = \bigcup_{y \in Y} K_y\ .$$

The set K_B of *base tokens* is

$$K_B = \bigcup_{y \in Y_B} K_y\ .$$

For each $k \in K$ and $y \in Y$, we say that k *has type* y iff

$$k \in K_y.$$

The terminal symbols of $\boldsymbol{X}$ are base tokens, the non-terminal ones are base types. A path in $\boldsymbol{X}$ from a base type y to a base token k means that k has type y, and a path from a base type y to another base type y' means that y is a super-type of y' (and symmetrically y' is a sub-type of y). The requirements that $\boldsymbol{X}$ is non-empty and has no

isolated symbols, assure that there are at least one base token and one base type, and that any base token has always at least one type. The acyclicity of $\boldsymbol{X}$ rules out edges from a base type to itself (which would make little sense), loops of two or more base types (which would indeed all contain the same tokens), and base types without base tokens (which might happen if there existed no path to any base token from any base type of a loop). Each string $y_1;\ldots;y_n$ of $n \geq 1$ base types is a type, whose tokens are all the strings $k_1;\ldots;k_n$ of base tokens of the corresponding base types[1]. We have organized tokens as a family of sets indexed by types. It is easy to see that if y is a sub-type of y', all tokens having type y also have type y', i.e. $K_y \subseteq K_{y'}$.

Note that, in principle, Definition 2 does not enforce user-defined types to be enumerative, in fact it does not prevent $\boldsymbol{X}$ from containing millions of symbols. Definition 2 must indeed be thought as accompanied by the informal requirement that it must be *practical* to explicitly draw the DAG (which hence must be "reasonably small"). Clearly, such a requirement cannot be formalized in a precise way.

In Figure 1 there is a base type Color with two sub-types Primary and Secondary. The base tokens of Primary are red, green, and blue, those of Secondary are cyan, magenta, and yellow[2]. For instance, the type Color;Color has sub-types Primary;Color, Secondary;Color, Primary;Primary, and so on. The token red;cyan has types Color;Color, Primary;Secondary, Color;Secondary, and Primary;Color.

2.3 Signature

As mentioned in Section 1, in SCP-nets the ways in which transitions can fire are defined by declarative expressions and guards on arcs and transitions. The possible expressions and guards which can be used in a given SCP-net are defined by the token taxonomy and by symbols denoting variables, functions, and predicates. Each variable has an associated type, and it can be only assigned tokens of that type. Each function has a pair of associated types, and maps tokens of the first type to tokens of the second type. Each predicate has an associated type, and maps tokens of that type to booleans. The concept of signature formally specifies the symbols denoting variables, functions, and predicates for an SCP-net.[3]

Definition 3. Given a token taxonomy $\boldsymbol{X}$, a *signature* for $\boldsymbol{X}$ is a triple

$$\boldsymbol{S} = \langle \{V_y\}_{y \in Y}, \{F_{y,y'}\}_{y,y' \in Y}, \{R_y\}_{y \in Y} \rangle$$

where:

(1) $\{V_y\}_{y \in Y}$ is a family of pairwise disjoint sets of symbols called *variables*;

(2) $\{F_{y,y'}\}_{y,y' \in Y}$ is a family of pairwise disjoint sets of symbols called *functions*;

(3) $\{R_y\}_{y \in Y}$ is a family of pairwise disjoint sets of symbols called *predicates*;

(4) $\left(\bigcup_{y \in Y} V_y \cup \bigcup_{y,y' \in Y} F_{y,y'} \cup \bigcup_{y \in Y} R_y \right)$ is finite.

The sets V, F, and R of all variables, functions, and predicates are thus

[1] Note that, for $n = 1$, each base type is also a type, and each base token is also a token, as expected.

[2] In additive color synthesis, red, green, and blue are primary, while cyan, magenta, and yellow are secondary. The situation is reversed in subtractive color synthesis.

[3] Note that in Definition 3 we refer, for example, to the set Y of types of the given token taxonomy $\boldsymbol{X}$, without explicitly stating that Y is the set of types of $\boldsymbol{X}$. In all the following definitions, we adopt this convention of referring to mathematical entities defined in previous definitions, by the same symbols used in those previous definitions (we have in fact chosen symbols in such a way to avoid any ambiguity).

$$V = \bigcup_{y \in Y} V_y \ , \quad F = \bigcup_{y,y' \in Y} F_{y,y'} \ , \quad \text{and} \quad R = \bigcup_{y \in Y} R_y \ .$$

Note that, for the sake of brevity, we terminologically identify variables, functions, and predicates with the symbols denoting them. The types associated to variables, functions, and predicates are defined by organizing them in families of pairwise disjoint sets, indexed by types or pairs of types.

In Figure 1 there are four variables (*p1*, *p2*, *s1*, and *s2*), two functions (*addsyn* and *subsyn*), and one predicate (*warm*), with the indicated associated types.

2.4 Interpretation

As we said in the previous subsection, each function of types y and y' maps tokens of type y to tokens of type y', and each predicate of type y maps tokens of type y to booleans. This is formally specified by the concept of interpretation, which attaches semantics to the function and predicate symbols of a signature.

Definition 4. Given a token taxonomy $\boldsymbol{X}$ and a signature $\boldsymbol{S}$ for $\boldsymbol{X}$, an *interpretation* for $\boldsymbol{X}$ and $\boldsymbol{S}$ is a pair

$$\boldsymbol{I} = \langle \phi, \rho \rangle$$

where:
(1) $\phi \in [\ F_{y,y'} \to [K_y \to K_{y'}]\ ]_{y,y' \in Y}$ and is called *function interpretation*;
(2) $\rho \in [\ R_y \to [K_y \to \mathbf{B}]\ ]_{y \in Y}$ and is called *predicate interpretation*.

In Figure 1, the semantics of *addsyn*, *subsyn*, and *warm* are shown in the tables. Substantially, *addsyn* and *subsyn* respectively represent additive synthesis of primary colors and subtractive synthesis of secondary colors, while *warm* returns T if its argument token represents a "warm" color (i.e. red, magenta, or yellow), F if it represents a "cold" color (i.e. green, blue, or cyan).

2.5 Expressions

Expressions are built out of tokens, variables, functions, and the same concatenation operator ; of tokens and types. This is formalized as follows.

Definition 5. Given a token taxonomy $\boldsymbol{X}$ and a signature $\boldsymbol{S}$ for $\boldsymbol{X}$, the family $\{E_y\}_{y \in Y}$ of *expressions* over $\boldsymbol{X}$ and $\boldsymbol{S}$ is the smallest family such that:
(1) $\forall\ y \in Y : K_y \cup V_y \subseteq E_y$;
(2) $\forall\ y, y' \in Y : \{\ f(e)\ \text{string} \mid f \in F_{y,y'} \wedge e \in E_y\ \} \subseteq E_{y'}$;
(3) $\forall\ y, y' \in Y : \{\ e;e'\ \text{string} \mid e \in E_y \wedge e' \in E_{y'}\ \} \subseteq E_{y;y'}$;
(4) $\forall\ y, y' \in Y$, y super-type of $y' : E_{y'} \subseteq E_y$.
The set E of all expressions is thus

$$E = \bigcup_{y \in Y} E_y \ .$$

For each $e \in E$ and $y \in Y$, we say that *e has type y* iff

$$e \in E_y .$$

We define the function *Var* over E as follows:
(1) $\forall\ k \in K : Var(k) = \varnothing$;
(2) $\forall\ v \in V : Var(v) = \{v\}$;
(3) $\forall\ f(e) \in E : Var(f(e)) = Var(e)$;
(4) $\forall\ e;e' \in E : Var(e;e') = Var(e) \cup Var(e')$.

Expressions are inductively defined as a family of sets, indexed by types. Any token of type y is also an expression of type y. Any variable of type y is also an expression of type y. Any string $f(e)$, where f is a function of types y and y', and e is an expression of type y, is an expression of type y'. Any concatenation $e_1;\ldots;e_n$ of $n \geq 1$ expressions $e_1, \ldots, e_n$ of types $y_1, \ldots, y_n$, is an expression of type $y_1;\ldots;y_n$. Furthermore, any expression of type y' also has type y, for any super-type y of y'.

The function *Var* collects all the variables occurring in an expression.

Examples of expressions for the SCP-net in Figure 1 are *p1*;*p2* (of types Color;Color, Primary;Color, etc.), *s1*;*addsyn*(*p2*;red) (of types Secondary;Color and Color;Color), and *subsyn*(cyan;yellow) (of type Color).

2.6 Guards

Guards are built out of expressions, predicates, two pre-defined logical connectives for negation and conjunction, and also a pre-defined logical constant for truth. This is formalized as follows.

Definition 6. Given a token taxonomy $\boldsymbol{X}$ and a signature $\boldsymbol{S}$ for $\boldsymbol{X}$, the set G of *guards* over $\boldsymbol{X}$ and $\boldsymbol{S}$ is the smallest set such that:
(1) $\texttt{TRUE} \in G$;
(2) $\forall\, y \in Y : \{\ r(e)\ \text{string} \mid r \in R_y \wedge e \in E_y\ \} \subseteq G$;
(3) $\{\ (\sim g)\ \text{string} \mid g \in G\ \} \subseteq G$;
(4) $\{\ (g_1\ \&\ g_2)\ \text{string} \mid g_1, g_2 \in G\ \} \subseteq G$.
We define the function *Var* over G as follows:
(1) $Var(\texttt{TRUE}) = \varnothing$;
(2) $\forall\, r(e) \in G : Var(r(e)) = Var(e)$;
(3) $\forall\, (\sim g) \in G : Var(\sim g) = Var(g)$;
(4) $\forall\, (g_1\ \&\ g_2) \in G : Var(g_1\ \&\ g_2) = Var(g_1) \cup Var(g_2)$.

Guards are inductively defined as a set. The symbol TRUE is a guard. Any string $r(e)$, where r is a predicate of type y, and e is an expression of type y, is a guard. Such guards can be combined by means of logical connectives ~ for negation and & for conjunction (which, as it is well-known, suffice to express any boolean combination).

The function *Var* collects all the variables occurring in a guard.

Examples of guards for the SCP-net in Figure 1 are *warm*(*p2*), (~ *warm*(yellow)), and (*warm*(*s2*) & TRUE).

2.7 Binding and Evaluation

If a token is assigned to each variable occurring in an expression or guard (with the token having the type of the variable), the expression evaluates to a token and the guard to a boolean. The assignment of tokens to variables is formalized by the concept of binding, and the extension of a binding over expressions and guards formalizes their evaluation.

Definition 7. Given a token taxonomy $\boldsymbol{X}$, a signature $\boldsymbol{S}$ for $\boldsymbol{X}$, and an interpretation $\boldsymbol{I}$ for $\boldsymbol{X}$ and $\boldsymbol{S}$, let $V' \subseteq V$ be a set of variables; a *binding* for V' is a function

$$\beta \in [\ (V_y \cap V') \rightarrow K_y\ ]_{y \in Y}.$$

Let $E' \subseteq E$ be the largest subset of E such that $\bigcup_{e \in E'} Var(e) \subseteq V'$; the *evaluation* of E' with β is defined as follows:

(1) $\forall\ k \in K : \beta(k) = k$;
(2) $\forall\ f(e) \in E' : \beta(f(e)) = \phi(f)(\beta(e))$;
(3) $\forall\ e;e' \in E' : \beta(e;e') = \beta(e);\beta(e')$.
Let $G' \subseteq G$ be the largest subset of G such that $\bigcup_{g \in G'} Var(g) \subseteq V'$; the *evaluation* of G' with β is defined as follows:
(1) $\beta(\texttt{TRUE}) = \mathsf{T}$;
(2) $\forall\ r(e) \in G' : \beta(r(e)) = \rho(r)(\beta(e))$;
(3) $\forall\ (\sim g) \in G' : \beta(\sim g) =$ **if** $(\beta(g) = \mathsf{F})$ **then** T **else** F;
(4) $\forall\ (g_1\ \&\ g_2) \in G' : \beta(g_1\ \&\ g_2) =$ **if** $(\beta(g_1) = \beta(g_2) = \mathsf{T})$ **then** T **else** F.

The evaluation of expressions makes use of the semantics of functions (specified in the interpretation), and implicitly attaches the semantics of concatenation to ; . The evaluation of guards makes use of the semantics of predicates (specified in the interpretation), and implicitly attaches the semantics of negation and conjunction to ~ and &, as well as the semantics of truth to TRUE.

For instance, for the SCP-net in Figure 1, if a binding β assigns green to *p2* and cyan to *s1*, the expression *s1*;*addsyn*(*p2*;red) evaluates to cyan;yellow, and the guard *warm*(*p2*) evaluates to F.

2.8 Token System

The concept of token system puts together a token taxonomy, a signature, and an interpretation.

Definition 8. A *token system* is a triple

$$\boldsymbol{K} = \langle \boldsymbol{X}, \boldsymbol{S}, \boldsymbol{I} \rangle$$

where:
(1) $\boldsymbol{X}$ is a token taxonomy;
(2) $\boldsymbol{S}$ is a signature for $\boldsymbol{X}$;
(3) $\boldsymbol{I}$ is an interpretation for $\boldsymbol{X}$ and $\boldsymbol{S}$.

2.9 Labeling

Each place of an SCP-net is labeled by a type. As we will see, each place can only be marked by (i.e. contain) tokens of its labeling type. Each arc is labeled by an expression having the type labeling the place which the arc connects, and each transition is labeled by a guard. As we will see, the possible ways in which a transition can fire are defined by the labeling expressions of the incoming and outgoing arcs and by the labeling guard.

Definition 9. Given a topology $\boldsymbol{T}$ and a token system $\boldsymbol{K}$, a *labeling* of $\boldsymbol{T}$ with $\boldsymbol{K}$ is a triple

$$\boldsymbol{L} = \langle \psi, \varepsilon, \gamma \rangle$$

where:
(1) $\psi \in [P \to Y]$ and is called *type function*;
(2) $\varepsilon \in [\ A.a \to E_{\psi(\chi_P(a))}\ ]$ and is called *expression function*;
(3) $\gamma \in [T \to G]$ and is called *guard function*.
For each $t \in T$, we define

$$Var(t) = \bigcup_{a \in In(t) \cup Out(t)} Var(\varepsilon(a)) \cup Var(\gamma(t)).$$

The function *Var* collects all the variables surrounding the transition, i.e. occurring in the guard or in the expression of some incoming or outgoing arc.

The types, expressions, and guards labeling places, arcs, and transitions in Figure 1 are indicated near the corresponding circles, arrows, and rectangles.

2.10 Simple Colored Petri Net

As expected, an SCP-net consists of a topology, a token system, and a labeling.

Definition 10. A *Simple Colored Petri Net* (*SCP-net*) is a triple

$$\boldsymbol{SCPN} = \langle \boldsymbol{T}, \boldsymbol{K}, \boldsymbol{L} \rangle$$

where:
(1) $\boldsymbol{T}$ is a topology;
(2) $\boldsymbol{K}$ is a token system;
(3) $\boldsymbol{L}$ is a labeling of $\boldsymbol{T}$ with $\boldsymbol{K}$.

2.11 Marking

Each place of an SCP-net can be marked by tokens of the type labeling the place. Multiple occurrences of a same token can mark a place.

Definition 11. Given an SCP-net ***SCPN***, a *marking* for ***SCPN*** is a function

$$\mu \in [\, P.p \rightarrow \mathcal{M}_\omega(K_{\psi(p)}) \,].$$

M is the set of all markings.

A marking associates a multiset of tokens to each place, instead of simply a set, to account for multiple occurrences of a same token.

The marking of the SCP-net in Figure 1 is indicated by the tokens inside places (base tokens are abbreviated for graphical reasons).

2.12 Enabling and Firing

Analogously to CP-nets, in an SCP-net a transition t can fire in as many ways as the possible bindings for the variables $Var(t)$, for which the guard evaluates to T. Given one such binding β, the firing of t with β consists of two steps. First, for each incoming arc of t, the token which the labeling expression evaluates to, is removed from the corresponding input place of t. Second, for each outgoing arc of t, the token which the labeling expression evaluates to, is added to the corresponding output place of t[4]. Of course, in order for the first step to take place, the tokens to be removed must be present in input places. These ideas are formalized as follows.

Definition 12. Given an SCP-net ***SCPN***, let:
(1) $\mu \in M$;
(2) $t \in T$;
(3) β be a binding for $Var(t)$.
We say that t is *enabled* with β in μ iff:
(1) $\forall\, p \in P : \bigcup_{a \in In(t) \cap Out(p)} \{\, \beta(\varepsilon(a)) \,\}_{\mathsf{m}} \subseteq \mu(p)$;
(2) $\beta(\gamma(t)) = \mathsf{T}$.

[4] Note that, if for another binding $\beta' \neq \beta$ all the arc expressions surrounding t evaluate to the same tokens as β, β' really specifies the same firing as β.

If t is enabled with β in μ, the marking μ' produced by *t firing* with β in μ is defined as follows:

$$\forall\, p \in P : \mu'(p) = \left(\mu(p) - \bigcup_{a \in In(t) \cap Out(p)} \{\, \beta(\varepsilon(a)) \,\}_{\mathsf{m}} \right) \cup \bigcup_{a \in Out(t) \cap In(p)} \{\, \beta(\varepsilon(a)) \,\}_{\mathsf{m}} .$$

We stress that the union, inclusion, and difference operators above operate on multisets, and not sets. In fact, for each place p, as many tokens (possibly none) as the arcs connecting p to t are removed from p, and as many tokens (possibly none) as the arcs connecting t to p are added to p.

For instance, in the SCP-net in Figure 1, consider the transition T2, whose surrounding variables in *Var*(T2) are *s1* and *s2*. If β is a binding assigning yellow to *s1* and magenta to *s2*, the guard evaluates to T, the expression labeling A2 evaluates to yellow;magenta, and the expression labeling A4 evaluates to red. Thus, T2 is enabled with β, and the firing of T2 with β amounts to removing yellow;magenta from P0 and adding red to P2.

3. Conclusions and Future Work

As it should be clear from the previous section, SCP-nets constitute a promising trade-off between P-nets and CP-nets, as claimed in Section 1. Note that, owing to the hierarchical organization of types, SCP-nets are not quite a sub-class of CP-nets (where any token or expression always has just one type). The type hierarchy indeed provides a lot of convenience. For example, a place can be enforced to contain only a subset of the tokens of a certain type, by means of a sub-type labeling the place (e.g. if in Figure 1 there were a place labeled by Primary, it could only contain primary color tokens). Without type hierarchy, the "sub-type" should be another type with duplicates of tokens of the "super-type", with the need of type conversions between the two (e.g. Primary would be another type with duplicates of red, green, and blue of type Color). As another example, different subsets of the tokens in a place can be enforced to be removed from the place when different output transitions fire, by labeling the outgoing arcs of the place by variables of sub-types of the type of the place (e.g. in Figure 1, since the type of *p1* and *p2* is Primary and the type of *s1* and *s2* is Secondary, when T1 fires only primary color tokens can be removed form P0, while when T2 fires only secondary color tokens can be removed form P0). Without type hierarchy, additional guards would be needed (e.g. the type of all of *p1*, *p2*, *s1*, and *s2* should be Color, and guards on T1 and T2 would be needed to require primary and secondary colors respectively), and therefore hierarchy allows clearer and more compact expressions and guards.

The main direction of future work consists in the study of methods for formal analysis and synthesis of SCP-nets. In particular, it would be interesting to try to generalize to SCP-nets the many known methods for P-nets.

Another interesting direction is the formal definition of hierarchical SCP-nets, which can recursively contain nested SCP-net fragments called *SCP-subnets* (similarly to hierarchical CP-nets). In addition, we would like to define *parameterized* SCP-subnets, i.e. collections of slightly different SCP-subnets which can be selected by means of actual values for parameters (e.g. an SCP-subnet representing a FIFO buffer, parameterized on the size and type of tokens to be stored). These formal definitions should serve as a foundation for the modular and re-usable development of SCP-nets. We have in fact planned to build software tools for the creation and editing of

SCP-nets, where libraries of parameterized SCP-subnets can be defined and used in different situations by suitably instantiating them.

Appendix

In this appendix we explain the mathematical concepts and notations used in Section 2.

$\mathbf{B} = \{\mathsf{T}, \mathsf{F}\}$ is the set of booleans.

$\mathbf{N} = \{0, 1, 2, \ldots\}$ is the set of natural numbers.

If $A_1, \ldots, A_n$ ($n \geq 2$) are sets, $A_1 \times \ldots \times A_n$ is the cartesian product of $A_1, \ldots, A_n$, i.e. the set of all n-tuples $\langle a_1, \ldots, a_n \rangle$ where $a_1 \in A_1, \ldots, a_n \in A_n$.

If A and B are sets, $[A \rightarrow B]$ is the set of all (total) functions with domain A and codomain B.

If A and B are sets, and $B(a) \subseteq B$ is a subset of B for each $a \in A$, $[A.a \rightarrow B(a)]$ is the set of all functions $f \in [A \rightarrow B]$ such that $f(a) \in B(a)$ for each $a \in A$.

If A and B are sets, and $\delta_a \in B$ is an element of B for each $a \in A$, $\{\delta_a\}_{a \in A}$ is the family of elements of B indexed by the elements of A, i.e. the function $f \in [A \rightarrow B]$ such that $f(a) = \delta_a$ for each $a \in A$. If $A = A' \times A'$ for some set A', we just write $\{\delta_{a_1,a_2}\}_{a_1,a_2 \in A'}$ instead of $\{\delta_{\langle a_1,a_2 \rangle}\}_{\langle a_1,a_2 \rangle \in A}$.

If $B = \{B_a\}_{a \in A}$ and $C = \{C_a\}_{a \in A}$ are families of pairwise disjoint sets, $[B_a \rightarrow C_a]_{a \in A}$ is the set of all functions $f \in \left[\bigcup_{a \in A} B_a \rightarrow \bigcup_{a \in A} C_a\right]$ such that $f(b) \in C_a$ for each $a \in A$ and $b \in B_a$.

If A is a set, $\mathcal{M}_\omega(A)$ is the set of all finite multisets over A, where a finite multiset over A is a function $ms \in [A \rightarrow \mathbf{N}]$ such that $\sum_{a \in A} ms(a) \in \mathbf{N}$. If $a \in A$, $ms(a)$ is the multiplicity of a in ms. If $a \in A$, $\{a\}_\mathsf{m}$ is the multiset $ms \in \mathcal{M}_\omega(A)$ defined by $ms(a) = 1$, and $ms(a') = 0$ for each $a' \in A - \{a\}$. If $ms_1, ms_2 \in \mathcal{M}_\omega(A)$, $ms_1 \cup ms_2$ is the multiset $ms \in \mathcal{M}_\omega(A)$ defined by $ms(a) = ms_1(a) + ms_2(a)$ for each $a \in A$. If $ms_1, ms_2 \in \mathcal{M}_\omega(A)$, we say that $ms_1 \subseteq ms_2$ iff $ms_1(a) \leq ms_2(a)$ for each $a \in A$. If $ms_1, ms_2 \in \mathcal{M}_\omega(A)$, and $ms_1 \subseteq ms_2$, $ms_2 - ms_1$ is the multiset $ms \in \mathcal{M}_\omega(A)$ defined by $ms(a) = ms_2(a) - ms_1(a)$ for each $a \in A$.

If A is a set, a directed graph of elements of A is a pair $\Gamma = \langle N, E \rangle$, where $N \subseteq A$ is the set of nodes, and $E \subseteq N \times N$ is the set of edges. Γ is empty iff $N = \emptyset$. Γ is finite iff N is finite. If $a, a' \in N$, we write $a \rightarrow_\Gamma a'$ to express that there are $a_1, \ldots, a_n \in N$ ($n \geq 0$) such that $\langle a, a_1 \rangle, \langle a_1, a_2 \rangle, \ldots, \langle a_n, a' \rangle \in E$ (i.e. there is a path from a to a' in Γ). Γ is acyclic iff no $a \in N$ is such that $a \rightarrow_\Gamma a$; we write "DAG" as a shortcut for "directed acyclic graph". A node $a \in N$ is terminal iff no $a' \in N$ is such that $a' \rightarrow_\Gamma a$ (note that if Γ is acyclic, there is at least one terminal node). Term(Γ) is the set of all terminal nodes of Γ, and NTerm(Γ) is the set of all non-terminal nodes of Γ (i.e. NTerm(Γ) = N − Term(Γ)). A node $a \in N$ is isolated iff no $a' \in N$ is such that $a' \rightarrow_\Gamma a$ or $a \rightarrow_\Gamma a'$ (note that an isolated node is also terminal).

References

[1] K. Jensen, "Coloured Petri Nets: A High Level Language for System Design and Analysis", in G. Rozenberg (ed.), *Advances in Petri nets 1990*, Lecture Notes in Computer Science, vol. 483, Springer-Verlag, pp. 342–416, 1990.

[2] T. Murata, "Petri Nets: Properties, Analysis and Applications", in *Proceedings*

of the IEEE, 77(4), pp. 541–580, April 1989.

[3] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[4] M. C. Zhou and F. DiCesare, *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems*, Kluwer Academic Publishers, 1993.