Chapter 9
An Introduction to Reflective Petri Nets

Lorenzo Capra
Università degli Studi di Milano, Italy

Walter Cazzola
Università degli Studi di Milano, Italy

ABSTRACT

Most discrete-event systems are subject to evolution during lifecycle. Evolution often implies the development of new features, and their integration in deployed systems. Taking evolution into account since the design phase therefore is mandatory. A common approach consists of hard-coding the foreseeable evolutions at the design level. Neglecting the obvious difficulties of this approach, we also get system's design polluted by details not concerning functionality, which hamper analysis, reuse and maintenance. Petri Nets, as a central formalism for discrete-event systems, are not exempt from pollution when facing evolution. Embedding evolution in Petri nets requires expertise, other than early knowledge of evolution. The complexity of resulting models is likely to affect the consolidated analysis algorithms for Petri nets. We introduce Reflective Petri nets, a formalism for dynamic discrete-event systems. Based on a reflective layout, in which functional aspects are separated from evolution, this model preserves the description effectiveness and the analysis capabilities of Petri nets. Reflective Petri nets are provided with timed state-transition semantics.

INTRODUCTION

Evolution is becoming a very hot topic in discrete-event system engineering. Most systems are subject to evolution during lifecycle. Think e.g. of mobile ad-hoc networks, adaptable software, business processes, and so on. Such systems need to be updated, or extended with new features, during lifecycle. Evolution can often imply a complete system redesign, the development of new features and their integration in deployed systems.

It is widely recognized that taking evolution into account since the system design phase should be considered mandatory, not only a good practice. The design of dynamic/adaptable discrete-event systems calls for adequate modeling formalisms and tools.

DOI: 10.4018/978-1-60566-774-4.ch009

Unfortunately, the known well-established formalisms for discrete-event systems lack features for naturally expressing possible run-time changes to system's structure.

System's evolution is almost always emulated by directly enriching original design information with aspects concerning possible evolutions. This approach has several drawbacks:

- all possible evolutions are not always foreseeable;
- functional design is polluted by details related to evolutionary design: formal models turn out to be confused and ambiguous since they do not represent a snapshot of the current system only;
- evolution is not really modeled, it is specified as a part of the behavior of the whole system, rather than an extension that *could* be used in different contexts;
- pollution hinders system's maintenance and reduces possibility of reuse.

Petri nets, for their static layout, suffer from these drawbacks as well when used to model adaptable discrete-event systems. The common modeling approach consists of merging the Petri net specifying the base structure of a dynamic system with information on its foreseeable evolutions. A similar approach pollutes the Petri net model with details not pertinent to the system's current configuration. Pollution not only makes Petri net models complex, hard to read and to manage, it also affects the powerful analysis techniques/tools that classical Petri nets are provided with.

System evolution is an aspect orthogonal to system behavior, that crosscuts both system deployment and design; hence it could be subject to separation of concerns (Hürsch & Videira Lopes, 1995), a concept traditionally developed in software engineering. Separating evolution from the rest of a system is worthwhile, because evolution is made independent of the evolving system and the above mentioned problems are overcome. Separation of concerns could be applied to a Petri net-based modeling approach as well. Design information (in our case, a Petri net modeling the system) will not be polluted by non pertinent details and will exclusively represent current system functionality without patches. This leads to simpler and cleaner models that can be analyzed without discriminating between what is and what could be system structure and behavior. Reflection (Maes, 1987) is one of the mechanisms that easily permits the separation of concerns.

Reflection is defined as the activity, both *introspection* and *intercession*, performed by an agent when doing computations about itself (Maes, 1987). A reflective system is layered in two or more levels (base-, meta-, meta-meta-level and so on) constituting a *reflective tower*; each layer is unaware of the above one(s). Base-level entities perform computations on the application domain entities whereas entities on the meta-level perform computations on the entities residing on the lower levels. Computational flow passes from a lower level (e.g., the base-level) to the adjacent level (e.g., the meta-level) by intercepting some events and specific computations (*shift-up action*) and backs when meta-computation has finished (*shift-down action*). All meta-computations are carried out on a representative of lower-level(s), called *reification*, which is kept *causally connected* to the original level. For details look at Cazzola, 1998.

Similarly to what is done in Cazzola, Ghoneim, & Saake, 2004, the meta-level can be programmed to evolve the base-level structure and behavior when necessary, without polluting it with extra information. In Capra & Cazzola, 2007 we apply the same idea to the Petri nets domain, defining a reflective Petri net model (hereafter referred to as Reflective Petri nets) that separates the Petri net describing a system from the high-level Petri net (Jensen & Rozenberg, 1991) that describes how this system evolves upon occurrence of some events/conditions. In this chapter we introduce Reflective Petri nets, and we propose a simple

state-transition semantics as a first step toward the implementation of a (performance-oriented) discrete-event simulation engine. With respect to other proposals recently appeared with similar goals (Cabac, Duvignau, Moldt, & Rölke, 2005; Hoffmann, Ehrig, & Mossakowski, 2005), Reflective Petri nets do not define a new Petri net paradigm, rather they rely upon a combination of consolidated classes of Petri nets and reflection concepts. What gives the possibility of using existing tools and analysis techniques in a fully orthogonal fashion. The short-time perspective is to integrate the GreatSPN graphical simulation environment (Chiola, Franceschinis, Gaeta, & Ribaudo, 1995) to directly support Reflective Petri nets.

In the rest of the chapter, we briefly present the (stochastic) Petri net classes used for the two levels of the reflective model; then we introduce Reflective Petri nets and the associated terminology, focusing on the (high-level) Petri net component (called *framework*) realizing the causal connection between the logical levels of the reflective layout; at last we provide a stochastic state-transition semantics for Reflective Petri nets; finally we present some related work and draw our conclusions and perspectives. An application of Reflective Petri nets to dynamic workflow design will be presented in the companion chapter (Capra & Cazzola, 2009).

SWN AND GSPN BASICS

Colored Petri nets (CPN) (Jensen & Rozenberg, 1991) are a major Petri net extension belonging to the high-level Petri nets category. For the *meta-level* of Reflective Petri nets we have chosen Well-formed Nets (WN) (Chiola, Dutheillet, Franceschinis, & Haddad, 1990), a CPN flavor (enriched with priorities and inhibitor arcs) retaining expressive power, characterized by a structured syntax. For performance analysis purposes, we are considering *Stochastic* Well-formed nets (SWN) (Chiola, Dutheillet, Franceschinis, & Haddad, 1993). SWN are the high-level counterpart of Generalized Stochastic Petri nets (GSPN) (Ajmone Marsan, Conte, & Balbo, 1984), the Petri net class used for the *base-level*. In other words, the unfolding of a SWN is defined in terms of a GSPN.

This section introduces SWN semi-formally, by an example. The GSPN definition is in large part derived. Figure 1 shows the portion of the evolutionary framework (Figure 3) that removes a given node from the base-level PN modeling the system (reified as a WN marking). The removal of a node provokes as side-effect the withdrawal of any arcs connected to the node itself. Trying to remove a marked place or a not-existing node cause a restart action. We assume hereafter that the reader has some familiarity with ordinary Petri nets.

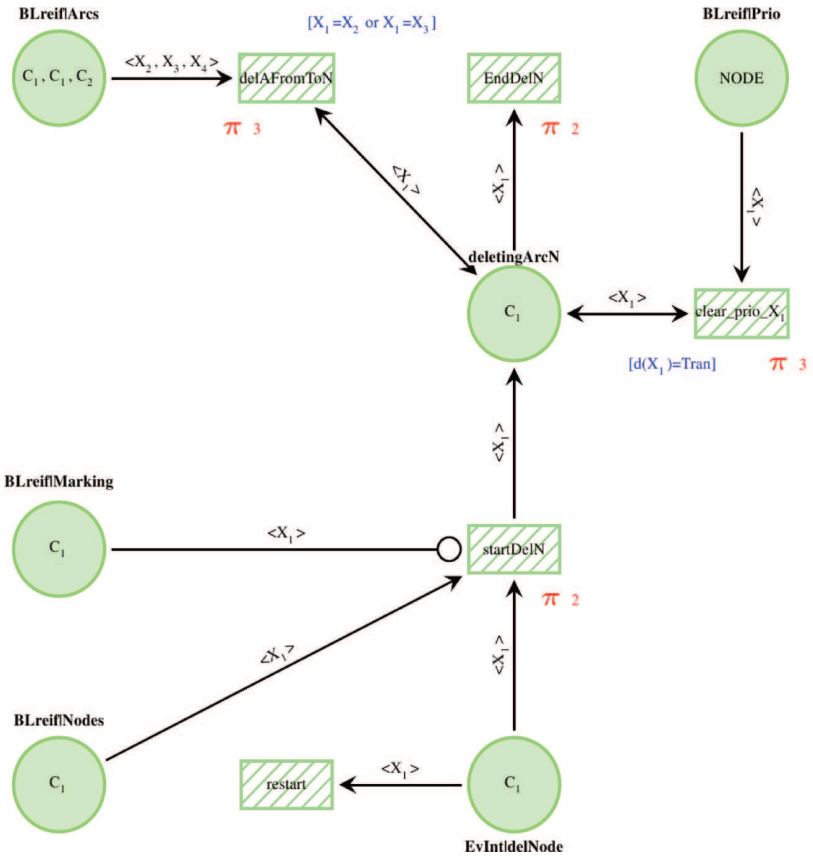
A SWN is a 11-tuple $(T, P, \{C_1, \ldots, C_n\}, \mathcal{C}, W^+, W^-, H, \mathrm{F}, \mathrm{P}, \mathbf{M}_0, \lambda)$

where P is the finite set of *places*, T is the finite set of *transitions*, $P \cap T = \varnothing$. With respect to ordinary Petri nets, places may contain ''colored'' tokens of different identity. $C_1, \ldots, C_n$ are finite basic color classes. In the example there are only two classes C_1, and C_2, denoting the base-level nodes, and the different kinds of connections between them, respectively. A basic color class may be partitioned in turn into static sub-classes, $C_i = \bigcup_k C_{i,k}$.

C assigns to each $s \in P \cup T$ a color domain, defined as Cartesian product of basic color classes: e.g. tokens staying at place BLreif|Arcs are triplets $\langle n_1, n_2, k_1 \rangle \in C_1 \times C_1 \times C_2$. A CPN transition actually folds together many elementary ones, so one speaks of instances of a colored transition. In Figure 1 $\mathcal{C}(t) = C_1$, for t ¹ delAFromToN ; $\mathcal{C}(\text{delAFromToN}) = C_1 \times C_1 \times C_1 \times C_2$. An instance of delAFromToN is thus a 4-tuple $\langle n_1, n_2, n_3, k_1 \rangle$.

A SWN marking **M** maps each place p to an element of $Bag(\mathcal{C}(p))$. $\mathbf{M_0}$ denotes the initial

Figure 1. A Well-Formed Net



marking.

W^-, W^+ and H assign each pair $(t, p) \in T \times P$ an (input, output and inhibitor, respectively) arc function $C(t) \rightarrow Bag(\mathcal{C}(p))$. Any arc function is formally expressed as a (linear combination of) *function-tuple(s)* $\langle f_1, \ldots, f_n \rangle$, tuple components are called *class-functions*. Each f_i is a function, $C(t) \rightarrow Bag(C_j)$, C_j being the color class on *i*-th position in $\mathcal{C}(p)$, and is called class-*j* function. Letting $F : \langle f_1, \ldots, f_n \rangle$ and $t_c : \langle c_1, \ldots, c_m \rangle \in \mathcal{C}(t)$, then $F(t_c) = f_1(t_c) \times \ldots f_n(t_c)$, where operator $\times$ denotes the multi-set Cartesian product. Each f_i is expressed in terms of *elementary* functions: the only ones appearing in this chapter are the projection X_k ($k \leq m$), defined as $X_k(t_c) = c_k$, and the constants S and $S_{j,k}$, mapping any t_c to C_j and $C_{j,k}$, respectively.

$\langle X_2, X_3, X_4 \rangle$ in Figure 1 (surrounding transition delAFromToN) is a function-tuple whose 1st and 2nd components are class-1 functions, while the 3rd one is a class-2 function: $\langle X_2, X_3, X_4 \rangle(\langle n_1, n_2, n_3, k_1 \rangle) = 1 \cdot n_2 \times 1 \cdot n_3 \times 1 \cdot k_1$, that is, $1 \cdot \langle n_2, n_3, k_1 \rangle$.

Φ associates a guard $[g] : \mathcal{C}(t) \rightarrow \{true, false\}$ to each transition t. A guard is built upon a set of basic predicates testing equality between projection applications, and membership to a given static subclass. As an example, $[X_1 = X_2 \vee X_1 = X_3](\langle n_1, n_2, n_1, k_1 \rangle) = true$.

A transition color instance $t_c \in \mathcal{C}(t)$ *has concession* in $\mathbf{M}$ if and only if, for each place p:

(i) $W^-(t, p)(t_c) \leq \mathbf{M}(p)$,

(ii) $H(t,p)(t_c) > \mathbf{M}(p)$,

(iii) $\Phi(t)(t_c) = true$

(the operators $>, £, +, -$ are here implicitly extended to multisets). $\Pi : T \to \mathbb{N}$ assigns a priority level to each transition. Level 0 is for *timed* transitions, while greater priorities are for *immediate* transitions, which fire in zero time.

t_c is *enabled* in **M** if it has concession, and no higher priority transition's instances have concession in **M**. It can fire, leading to **M'**:

$$\forall p \in P\ \mathbf{M}'(p) = \mathbf{M}(p) + W^+(t,p)(t_c) - W^-(t,p)(t_c)$$

Finally, $\lambda : T \to \mathbb{R}^+$ assigns a rate, characterizing an exponential firing delay, to each timed transition, and a weight to each immediate transition. Weights are used to probabilistically solve conflicts between immediate transitions with equal priority.

The behavior of a SWN model is formally described by a state-transition graph (or reachability-graph), which is built starting from $\mathbf{M}_0$. As a result of the SWN time representation, the SWN reduced reachability graph, which is obtained by suitably removing those markings (called vanishing) enabling some immediate transitions, is isomorphic to a Continuous Time Markov Chain (CTMC) (Chiola, Dutheillet, Franceschinis, & Haddad, 1993).

Special *restart transitions*, denoted by prefix *rest*, are used in our models once again for modeling convenience (we might always trace it back to the standard SWN definition). While the enabling rule of restart transitions doesn't change, their firing leads a SWN model back to the initial marking.

The class of Petri nets used for the *base-level* correspond to the unfolded (uncolored) version of SWN, that is, GSPN (Ajmone Marsan, Conte, & Balbo, 1984). A GSPN is formally a 8-tuple $(T, P, W^+, W^-, H, \Pi, \mathbf{m}_0, \lambda)$.

With respect to SWN definition, W^+, W^-, H are functions $T \times P \to \mathbb{N}$. Analogously, a marking **m** is a mapping $P \to \mathbb{N}$. The definitions of concession, enabling, firing given before are still valid (guards have disappeared), but for replacing $F(t,p)(t_c)$ by $F(t, p)$, and interpreting the operators in the usual way.

SWN Symbolic Marking Notion

The particular syntax of SWN color annotations allows system symmetries to be implicitly embedded into SWN models. This way efficient algorithms can be applied, e.g., to build a compact Symbolic Reachability Graph (SRG) (Chiola, Dutheillet, Franceschinis, & Haddad, 1997), with an associated *lumped* CTMC, or to launch symbolic discrete-event simulation runs. These algorithms rely upon the notion of *Symbolic Marking* (SM).

A SM provides a *syntactical* equivalence relation on ordinary SWN colored markings: two markings belong to the same SM if and only if they can be obtained from one another by means of *permutations* on color classes that preserve static subclasses.

Formally, a SM $\hat{\mathbf{M}}$ comprises two parts specifying the so called dynamic subclasses and the distribution of colored symbolic tokens (tuples built of dynamic subclasses) over places, respectively. Dynamic subclasses define a *parametric partition* of color classes preserving static subclasses: let $\hat{C}_i$ and s_i denote the set of dynamic subclass of C_i (in a given $\hat{\mathbf{M}}$), and the number of static subclasses of C_i. The *j*-th dynamic subclass $Z_j^i \in \hat{C}_i$ refers to a static subclass, denoted $d(Z_j^i)$, $1 \le d(Z_j^i) \le s_i$, and has an associated cardinality $|Z_j^i|$, i.e., it represents a parametric set of colors (we shall consider cardinality one dynamic subclasses). It must hold:

$$\forall k : 1 \ldots s_i \quad \sum_{j:d(Z_j^i)=k} |Z_j^i| = |C_{i,k}|.$$

The token distribution in $\hat{\mathbf{M}}$ is defined by a

function mapping each place p to a multiset on the *symbolic* color domain of p, $\hat{C}(p)$, obtained replacing C_i with $\hat{C}_i$ in $C(p)$.

Among several, possible equivalent forms, the SM canonical representative (Chiola, Dutheillet, Franceschinis, & Haddad, 1997) provides a univocal representation for SM, based on a lexicographic ordering of dynamic subclass distribution over places.

REFLECTIVE PETRI NETS

The *Reflective Petri nets* approach (Capra & Cazzola, 2007) quite strictly adheres to the classical reflective paradigm (Cazzola, 1998). It permits anyone having a basic knowledge of ordinary Petri nets to model a system and *separately* its possible evolutions, and to dynamically adapt system's model when evolution occurs.

The adopted reflective architecture (sketched in Figure 2) is structured in two logical layers. The first layer, called *base-level PN*, is represented by the GSPN specifying the system prone to evolve; whereas the second layer, called *meta-level* is represented by the *evolutionary meta-program*; in our case the meta-program is a SWN composed by the *evolutionary strategies*, which might drive the evolution of the base-level PN. More precisely, in the description below we will refer to the (untimed) carriers of SWN (i.e., WN nets) and GSPN, respectively, according to (Capra & Cazzola, 2007). Considering also the stochastic extension is straightforward, as discussed at the end of the next sub-section.

We realistically assume that several strategies

Figure 2. A Snapshot of the Reflective Layout

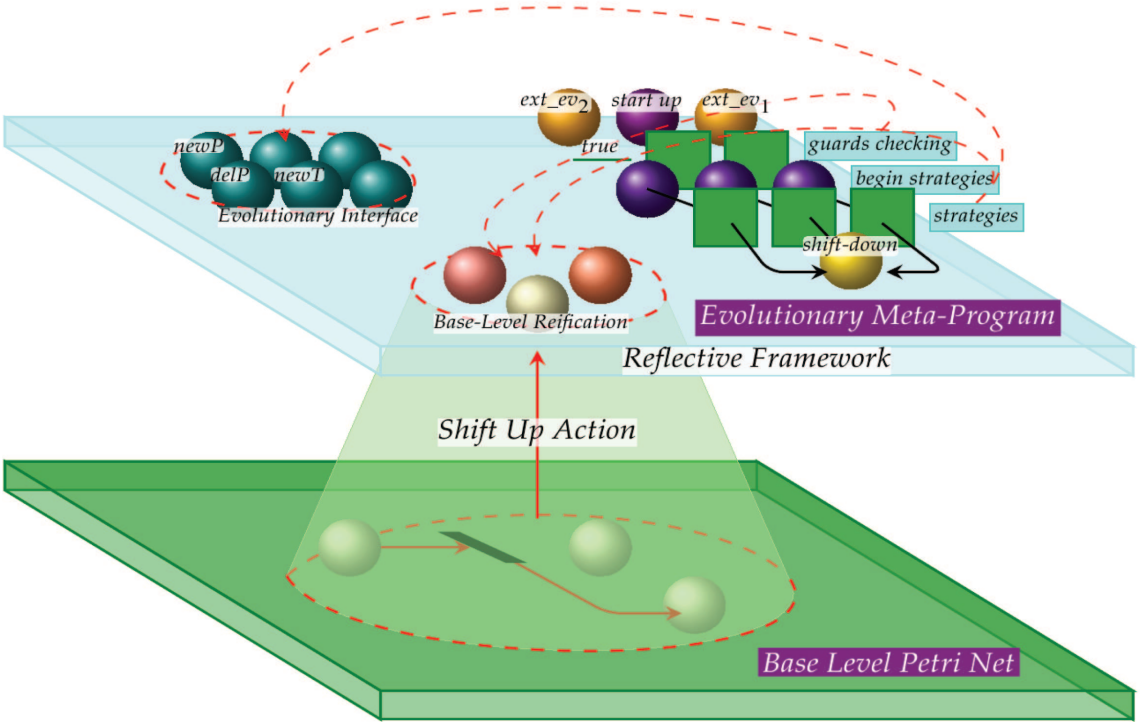
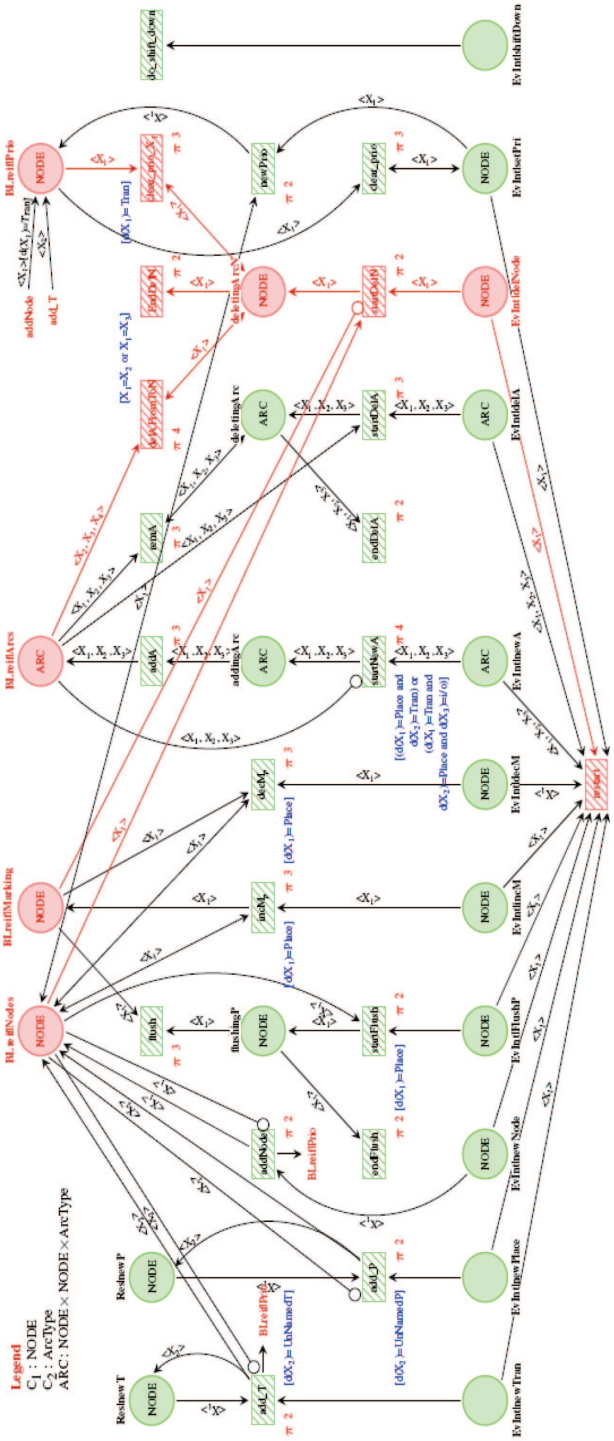


Figure 3. A Detailed View of the Framework Implementing the Evolutionary Interface



are possible at a given instant: in such a case one is selected in non-deterministic way (default policy). Evolutionary strategies have a *transactional* semantics: either they succeed, or leave the base-level PN unchanged.

The *reflective framework*, realized by a WN as well, is responsible for really carrying out the evolution of the base-level PN. It reifies the base-level PN into the meta-level as colored *marking* of a subset of places, called *base-level reification*, with some analogy to what is proposed in Valk, 1998. The base-level reification is updated every time the base-level PN enters a new state, and is used by the evolutionary meta-program to observe (*introspection*) and manipulate (*intercession*) the base-level PN. Each change to the reification will be reflected on the base-level PN at the end of a meta-program iteration, i.e., the base-level PN and its reification are *causally connected* and the reflective framework is responsible for maintaining this connection.

According to the reflective paradigm, the base-level PN runs irrespective of the evolutionary meta-program. The evolutionary meta-program is activated (*shift-up action*), i.e., a suiTable strategy is put into action, under two conditions non mutually exclusive: i) when triggered by an external event, and/or ii) when the base-level PN model reaches a given configuration.

Intercession on the base-level PN is carried out in terms of basic operations on the base-level reification suggested by the evolutionary strategy, called *evolutionary interface*, which permit any kind of evolution regarding both the structure and the current state (marking) of the base-level PN.

Each evolutionary strategy works on a specific area of the base-level PN, called *area of influence*. A conflict could raise when the changes induced by the selected strategy are reflected back (*shift-down action*) on the base-level, since influence area's local state could vary, irrespective of meta-program execution. To avoid possible inconsistency, the strategy must explicitly preserve the state (marking) of this area during its execution. To this aim the base-level execution is *temporary* suspended (using priority levels) until the reflective framework has inhibited any changes to the influence area of the selected evolutionary strategy. The base-level PN afterward resumes. This approach would favor concurrency between levels, and in perspective, between evolutionary strategies as well.

The whole reflective architecture is characterized by a fixed part (the reflective framework WN), and by a part varying from time to time (the base-level PN and the WN representing the meta-program). The framework hides evolutionary aspects to the base-level PN. This approach permits a clean separation between evolutionary model and evolving system model (look at the companion chapter (Capra & Cazzola, 2009) for seeing the benefits), which is updated only when necessary. So analysis/validation can be carried out separately on either models, without any pollution.

Reflective Framework

The framework formalization in terms of (S)WN allows us to specify complex evolutionary patterns for the base-level PN in a simple, unambiguous way.

The reflective framework (Figure 3) driven on the content of the evolutionary interface performs a sort of concurrent-rewriting on the base-level PN, suitably reified as a WN marking.

Places with prefix "BLreif"[1] belong to the base-level reification (*BLreif*), while those having prefix "EvInt" belong to the evolutionary interface (*EvInt*). Both categories of places represent interfaces to the evolutionary strategy sub-model.

While topology and annotations (color domains, arc functions, and guards) of the framework are fixed and generic, the structure of basic color classes and the initial marking need to be instantiated for setting a link between meta-level and base-level. In some sense they are similar to

formal parameters, which are bound to a given base-level PN.

Let $\mathcal{BL}_0 : (P_0, T_0, W_0^+, W_0^-, H_0, \Pi_0, \mathbf{m}_0)$ be the base-level PN at system start-up. The framework basic color classes are C_1: *NODE*, C_2: *ArcType*. We have **Definition 1**, where $P_0 \subseteq Place, T_0 \subseteq Tran$.

Class *ArcType* identifies two types of WN arcs, input/output and inhibitor. Class *NODE* collects the potential nodes of any base-level PN evolutions, therefore it should be set large enough to be considered as a logically unbounded repository. The above partitioning of *NODE* into singleton static subclasses may be considered as a default choice, which might be further adapted, depending on modeling/analysis needs. Symbols $p_i(t_j)$ denote base-level places (transitions) that can be explicitly referred to in a given evolutionary strategy. Instead symbols $x_i(y_j)$ denote anonymous places (transitions) added from time to time to the base-level without being explicitly named. To make it possible the automatic updating of the base-level reification (as explained in the sequel), also these elements can be referred to, but only at the net level, by means of WN constant functions.

Base-Level Reification

The color domains for the base-level PN reification are given below.

Definition 2 (Reification Color Domains)

$$\begin{array}{lll} \mathcal{C}(p) & : & NODE \quad \forall p \in BLreif \setminus \{\text{BLreif} \mid \text{Arcs}\} \\ \mathcal{C}(\text{BLreif} \mid \text{Arcs}) & : & ARC = NODE \times NODE \times ArcType \end{array}$$

The reification of the base-level into the framework, i.e., its encoding as a WN marking, takes place at system start-up (initialization of the reification), and just after the firing of any base-level transition, when the current reification is updated.

Definition 3 (reification marking) *The reification of Petri net* $\mathcal{BL} : (P, T, W^+, W^-, H, \Pi, \mathbf{m}_0)$, $reif(\mathcal{BL})$, is the marking:

$$\begin{aligned} \mathbf{M}(\text{BLreif} \mid \text{Nodes}) &= \sum_{n \in P \cup T} 1 \cdot n \\ \mathbf{M}(\text{BLreif} \mid \text{Prio}) &= \sum_{t \in T} (\Pi(t) + 1) \cdot t \\ \mathbf{M}(\text{BLreif} \mid \text{Marking}) &= \sum_{p \in P} \mathbf{m}_0(p) \cdot p \end{aligned}$$

$$\forall p \in P, t \in T \begin{cases} \mathbf{M}(\text{BLreif} \mid \text{Arcs})(\langle p, t, i/o \rangle) = W^-(p,t) \\ \mathbf{M}(\text{BLreif} \mid \text{Arcs})(\langle t, p, i/o \rangle) = W^+(p,t) \\ \mathbf{M}(\text{BLreif} \mid \text{Arcs})(\langle p, t, h \rangle) = H(p,t) \\ \mathbf{M}(\text{BLreif} \mid \text{Arcs})(\langle t, p, h \rangle) = 0 \end{cases}$$

The evolutionary framework's colored initial marking ($\mathbf{M}_0$) is the reification of base-level PN at system start-up ($reif(\mathcal{BL}_0)$). Place BLreif|Nodes holds the set of base-level nodes; the marking of place BLreif|Arcs encodes the connections between them: the term $2\langle t_2, p_1, i/o \rangle$ corresponds to an output arc of weight 2 from transition t_2 to place p_1.

Transition priorities are defined by the marking of BLreif|Prio: if t_2 is associated to priority level k, there will be the term $(k+1) \cdot \langle t_2 \rangle$ in BLreif|Prio. The above three places represent the base-level topology: any change operated by the

Definition 1. (Basic Color Classes)

$$\begin{aligned} ArcType &= i/o \cup h \\ NODE &= \underbrace{\overbrace{p_1 \cup p_2 \ldots}^{Named_p} \cup \overbrace{x_1 \cup x_2 \ldots}^{Unnamed_p}}_{Place} \bigcup \underbrace{\overbrace{t_1 \cup t_2 \ldots}^{Named_t} \cup \overbrace{y_1 \cup y_2 \ldots}^{Unnamed_t}}_{Tran} \cup \; null \end{aligned}$$

evolutionary strategy to their marking causes a change to the base-level PN structure that will be reflected at any shift-down from the meta-level to the base-level.

The marking of place BLreif|Marking defines the base-level (current) state: the multiset $2\langle p_1\rangle + 3\langle p_2\rangle$ represents a base-level marking where places p_1 and p_2 hold two and three tokens, respectively. At the beginning BLreif|Marking holds the base-level initial state.

The marking of BLreif|Marking can be modified by the evolutionary strategy itself, causing a real change to the base-level current state immediately after the shift-down action.

Conflicts and inconsistencies due to the concurrent execution of several strategies are avoided by defining an influence area for each strategy; such an influence area delimits a critical region that can be accessed only by one strategy at a time. More details on the influence areas are in the section about the model semantics.

The meaning of each element of the *BLreif* interface is summarized in Table 1. Let us only remark that some places of the interface (e.g. BLreif|Arcs) hold multisets, while other (e.g. BLreif|Nodes) logically hold only sets (in such a case the reflective framework is in charge of eliminating duplicates).

As subject to change, the base-level reification needs to preserve a kind of well-definiteness over the time. Let $\overline{m}$ be the support of multiset m, i.e., the set of elements occurring on m.

Definition 4 (well-defined marking) *Let* n_1, n_2: *NODE*, k: *ArcType*. **M** is well-defined if and only if

- $\overline{\mathbf{M}(\text{BLreif} \mid \text{Marking})} \subseteq Place \cap \overline{\mathbf{M}(\text{BLreif} \mid \text{Nodes})}$
- $\overline{\mathbf{M}(\text{BLreif} \mid \text{Prio})} \equiv Tran \cap \overline{\mathbf{M}(\text{BLreif} \mid \text{Nodes})}$
- if n_1 occur on $\overline{\mathbf{M}(\text{BLreif} \mid \text{Arcs})}$ then $n_1 \in \overline{\mathbf{M}(\text{BLreif} \mid \text{Nodes})}$
- $\langle n_1, n_2, k\rangle \in \mathbf{M}(\text{BLreif} \mid \text{Arcs}) \Rightarrow$ $\langle n_1, n_2\rangle \in Place \times Tran \vee$ $\langle n_1, n_2\rangle \in Tran \times Place \wedge k = i/o$

The other way round, a well-defined WN marking provides a univocal representation for the base-level PN.

Definition 5 (base-level mapping) *The GSPN* $bl(\mathbf{M}) : (P, T, W^+, W^-, H, \Pi, \mathbf{m}_0)$, associated to a well-defined **M**, is such that: $P = Place \cap \overline{\mathbf{M}(\text{BLreif} \mid \text{Nodes})}$, $T = Tran \cap \overline{\mathbf{M}(\text{BLreif} \mid \text{Nodes})}$, $\forall p \in P$ $\mathbf{m}_0(p) = \mathbf{M}(\text{BLreif} \mid \text{Marking})(p)$, $\forall t \in T$ $\Pi(t) = \mathbf{M}(\text{BLreif} \mid \text{Prio})(t) - 1$, finally W^-, W^+, H are set as in Definition 3 (reading equations from right to left).

From definitions above it directly follows $bl(reif(\mathsf{BL})) = \mathsf{BL}$. By the way $\mathbf{M}_0$ is assumed well-defined. Through the algebraic structural calculus for WN introduced in Capra, De Pierro, & Franceschinis, 2005 it has been verified that well-definiteness is an invariant of the evolutionary framework (Figure 3), and consequently of the whole reflective model. The proof, involving a lot of technicalities, is omitted.

Including the time information of GSPN and SWN in the reflective model is immediate, once we restrict to integer values for transition rate/weights (as if λ where a mapping $T \to \mathbb{N}^+$). The encoding of transition parameters then would be analogous to transition priorities. The *BLreif* interface (and of course also *EvInt*) would include an additional place BLreif|Param (EvInt| Param), with domain *NODE*. A base-level transition t_1 with firing rate k would be reified by a token $k \cdot \langle t_1\rangle$ on place BLreif|Param.

Evolutionary Framework Behavior

The evolutionary framework WN model implements a set of basic transformations (rewritings) on the base-level PN reification. Its structure is modular, being formed by independent subnets (easily recognizable) sharing interface *BLreif*, each implementing a basic transformation.

The behavior associated to the evolutionary framework is intuitive. Every place labeled by the EvInt prefix holds a (set of) basic transformation

Table 1. The Evolutionary Interface API and the Base-Level Reification Data Structure

Evolutionary Interface (the asterisk means that the marking is a set)	
EvInt| newTran* adds an anonymous transition to the base-level reification.	EvInt| newPlace* adds an anonymous place to the base-level reification.
EvInt| newNode* adds a given new node in the base-level reification.	EvInt| FlushP* flushes out the current marking of a place in the base-level reification.
EvInt| IncM increments the marking of a place in the base-level.	EvInt| decM decrements the marking of a place in the base-level.
EvInt| newA adds a new arc between a place and a transition in the base-level reification.	EvInt| delA deletes an arc between a place and a transition in the base-level reification.
EvInt| delNode* deletes a given node in the base-level reification (places must be empty).	EvInt| setPrio changes the priority to a node in the base-level reification.
EvInt| shiftDown* instructs the framework to reflect the changes on the base-level.	
Reification (the asterisk means that the marking is a set)	
BLreif| Nodes* the content of this place represents the nodes of the base-level PN.	BLreif| Marking the content of this place represents the current marking of the base-level PN.
BLreif| Arcs the content of this place represents the arcs of the base-level PN.	BLreif|Prio the content of this place represents the transition priorities of the base-level PN.

command(s) issued by the evolutionary strategy sub-model. Every time a (multiset of) token(s) is put on one of these places, a sequence of immediate transitions implementing the corresponding command(s) is triggered. A succeeding command results in changing the base-level reification, that is, the marking of *BLreif* places.

The implemented basic transformations are: adding/removing given nodes (EvInt|newNode, EvInt|delNode), adding anonymous nodes (EvInt|newPlace, EvInt|newTran), adding/removing given arcs (EvInt|newA, EvInt|delA), increasing/decreasing the marking of given places (EvInt|incM, EvInt|decM), flushing tokens out from places (EvInt|FlushP), finally, setting the priority of transitions (EvInt|setPrio). The color domain of each place (either *NODE* or *ARC*) corresponds to the type of command argument, except for EvInt|newPlace, EvInt|newTran, which are uncolored places.

Term $2\langle p_1 \rangle$ occurring on place EvInt|incM is interpreted as ''increase the current marking of place p_1 of two units''. Many commands of the same kind can be issued simultaneously, e.g. $2\langle p_1 \rangle + 1\langle p_3 \rangle$ on EvInt|incM. Depending on their meaning, some commands are encoded by multisets (as in the last examples), while other are encoded by sets. Interface *EvInt* is described on Table 1 and is implemented by the net on Figure 3.

In some cases command execution result must be returned back: places whose prefix is Res hold command execution results, e.g., places Res|newP and Res|newT record references to the last nodes that have been added to the base-level reification anonymously. Initially they hold a *null* reference. As interface places, they can be acceded by the evolutionary strategy sub-model.

Single commands are carried out in *consistent* and *atomic* way, and they may have side effects.

Let us consider for instance deletion of an existing node, which is implemented by the subnet depicted (in isolation) in Figure 1. Assume that a token n_1 is put in place EvInt|delNode. First the membership of n_1 to the set of nodes currently reified as not marked is checked (transition startDelN). In case of positive check the node is removed, then all surrounding arcs are removed (transition delAfromToN), last (if n_1 is a transition) its priority is cleared (transition clearPrio$_{x1}$). Otherwise the command aborts and the whole meta-model composed by the reflective framework and the evolutionary strategy is restarted, ensuring a transactional execution of the evolutionary strategy. A unique restart transition appears in Figure 3, with input arcs having an ''OR'' semantics.

Different priority levels are used to guarantee the correct firing sequence, also in case of many deletion requests (tokens) present in EvInt|delNode simultaneously. Boundedness is guaranteed by the fact that each token put on this place is eventually consumed.

The other basic commands are implemented in a similar way. Let us only remark that newly introduced base-level transitions are associated to the default priority 0 (encoded as 1).

Priority levels in Figure 3 are *relative*: after composing the evolutionary framework WN model to the evolutionary strategy WN model, the minimum priority in the evolutionary framework is set greater than the maximum priority level used in the evolutionary strategy.

Any kind of transformation can be defined as a combination of basic commands: for example ''replacing the input arc connecting nodes *p* and *t* by an inhibitor arc of cardinality three'' corresponds to put the token $\langle p,t,i/o\rangle$ on EvInt|delA and the term $3\langle p,t,h\rangle$ on place EvInt|newA. Who designs a strategy (the meta-programmer) is responsible for specifying consistent sequences of basic commands, e.g., he/she must take care of flushing the contents of a given place before removing it.

Base-level Introspection. The evolutionary framework includes basic introspection commands. Observation and manipulation of base-level PN reification are performed passing through the framework evolutionary interface; what enhances safeness and robustness of evolutionary programming. Figure 4 shows (from left to right) the subnets implementing the computation of the cardinality (thereupon the kind) of a given arc, the preset of a given base-level node, and the current marking of a given place (subnets computing transition priorities, post-sets, inhibitor-sets, and checking existence of nodes, have a similar structure).

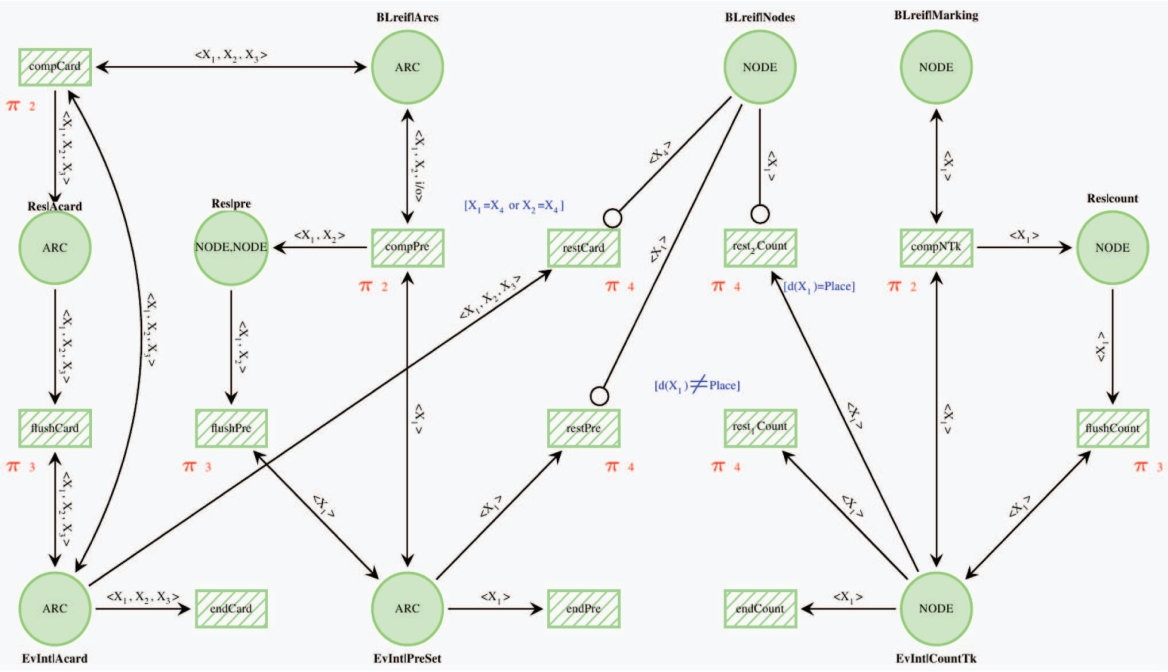
As for the basic transformation commands, each subnet has a single entry-place belonging to the evolutionary interface *EvInt* and performs atomically. Introspection result is recorded on places having the Res| prefix, accessible by the evolutionary strategy: regarding e.g., preset computation, a possible result (after a token p_1 has been put in place EvInt|PreSet) is $\langle p_1,t_2\rangle+\langle p_1,t_3\rangle$, meaning the preset of p_1 is $\{t_2,t_3\}$ (other results are encoded as multisets). Since base-level reification could be changed in the meanwhile, every time a new command is issued any previously recorded result about command's argument is cleared (transitions prefixed by string "flush").

The Evolutionary Strategy

The adopted model of evolutionary strategy (only highlighted in Figure 2) specifies a set of arbitrarily complex, alternative transformation patterns on the base-level (each denoted hereafter as *i*-th strategy or st_i), which can be fired when some conditions (checked on the base-level PN reification by introspection) hold and/or some external events occur.

Since a strategy designer is usually unaware of the details about the WN formalism, we have provided him/her with a tiny language that allows everyone to specify his own strategy in a simple and formal way. As concerns control structures the lan-

Figure 4. Basic introspection functions



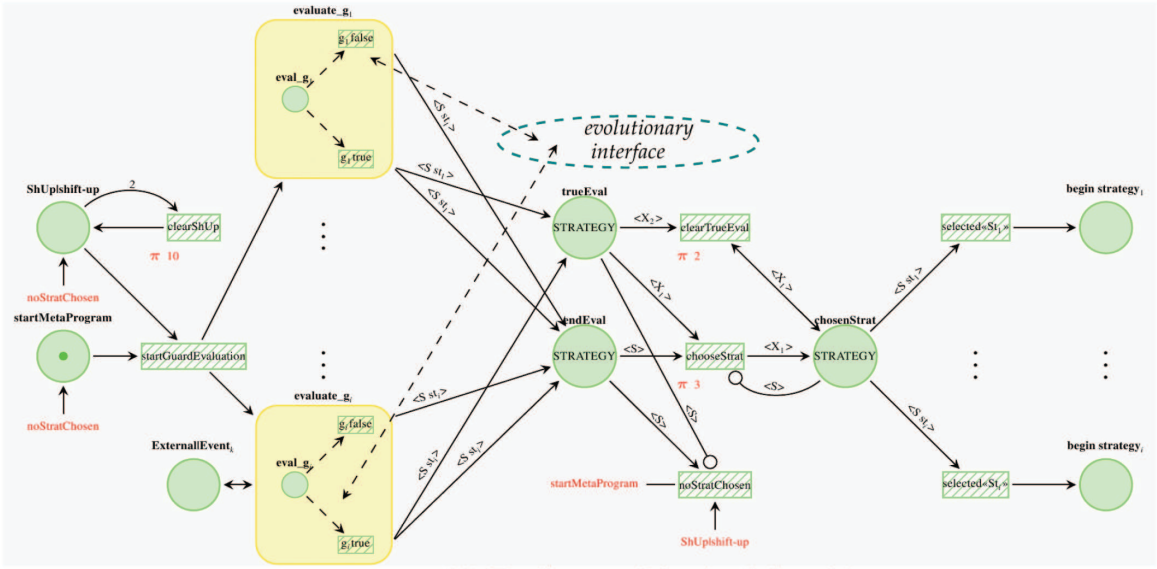
guage syntax is inspired by Hoare's CSP (Hoare, 1985), enriched with a few specific notations. As concerns data types, a basic set of built-in's and constructors is provided for easy manipulation of nets. The use of a CSP-like language to specify a strategy allows its automatic translation into a corresponding WN model. We will provide some examples of mapping from pieces of textual strategy descriptions into corresponding WN models. In Petri nets literature there are lot of examples of formal mappings from CSP-like formalisms (e.g. process algebras) to (HL)Petri nets models (e.g. Best, 1986 and more recently Kavi, Sheldon, Shirazi, & Hurson, 1995), from which we have been inspired.

The evolutionary meta-program scheme corresponds to the CSP pseudo-code[2] in Figure 6. The evolutionary strategy as a whole is cyclically activated upon a shift-up, here modeled as an input command. A non-deterministic selection of guarded commands then takes place. Each guard is evaluated on base-level reification by using "ad-hoc" language notations described in the sequel. Guard *true* means the corresponding strategy may be always activated at every shift-up. A guard optionally ends with an input command simulating the occurrence of some external events.

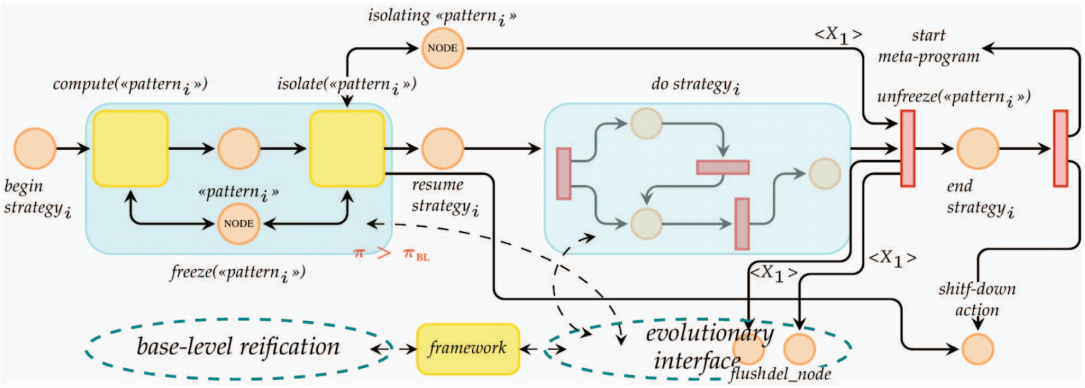
A more detailed view of this general schema in terms of Petri nets is given in Figure 5. Figure 5(a) shows the non-deterministic selection, whereas Figure 5(b) shows the structure of *i*-th strategy. Color domain definitions are inherited from the evolutionary framework WN. An additional basic color class ($STRAT = st_1 \cup \ldots st_n$) represents possible alternative evolutions

Focusing on Figure 5(a), we can observe that any shift-up is signaled by a token in the homonym place, and guards (the boxes on the picture, which represent the only not fixed parts of the net) are evaluated concurrently, accordingly to the semantics of CSP alternative command. After the evaluation process has been completed one branch (i.e., a particular strategy) is chosen (transition chooseStrat) among those whose guard

Figure 5. Meta-Program Generic Schema



(a) The Strategy Selection Submodel



(b) The Strategy Structure

was successfully evaluated (place trueEval). By the way, introspection has to be performed with priority over base-level activities, so the lowest priority in Figure 5(a) is set higher than any base-level PN transition, when the whole model is built. In case every guard is valued false the selection command is restarted just after a new shift-up occurrence transition noStratChoosen), avoiding any possible *livelock*. Occurrence of external events is modeled by putting tokens in particular ‘‘open’‘ places (e.g. External|event$_k$ in Figure 5(a). The idea is that such places should be shared with other sub-models simulating the external event occurrence. If one is simply interested in interactively simulating the reflective model, he/she might think of such places as a sort of buttons to be pushed by request.

(a) The Strategy Selection Submodel
(b) The Strategy Structure

Figure 6. CSP code for the meta-program scheme

```
*[shift-up ? sh-up-occurred →
     [
      guard_1; event_1 ? event_1-occurred → strategy_1()
        □
      guard_2 → strategy_2()
        □
      true → strategy_3()
        □
      ...
      ]
 ]
```

The i^{th} Strategy. The structure of the WN model implementing a particular evolutionary strategy is illustrated in Figure 5(b). It is composed of fixed and programmable (variable) parts, which may be easily recognized in the picture. It realizes a sort of *two-phases* approach: during the first phase (subnet freeze(«pattern$_i$»)) the meta-program sets the local influence area of the strategy, a portion of the base-level Petri Net reification potentially subject to changes. This area is expressed as a language's "pattern", that is, a parametric set of base-level nodes defined through the language notations, denoted by a colored homonym place in Figure 5(b). The pattern contents are flushed at any strategy activation. A simple isolation algorithm is then executed, which freezes the strategy influence area reification, followed by a shift-down action as a result of which freezing materializes at the base-level PN. The idea is that all transitions belonging to the pattern, and/or able to change the marking of places belonging to it, are temporary inhibited from firing, until the strategy execution has terminated (the place pattern* holds a wider pattern image after this computation).

```
*[shift-up ? sh-up-occurred →
[
guard_1; event_1 ? event_1-occurred →
strategy_1()
  □
guard_2 → strategy_2()
  □
true → strategy_3()
  □
...
]
]
```

During the freezing phase the base-level model is "suspended" to avoid otherwise possible inconsistencies and conflicts: this is achieved by forcing transitions of freeze(«pattern$_i$») subnet to have a higher priority than base-level PN transitions. The freeze(«pattern$_i$») sub-model is decomposed in turn in two sub-models that implement the influence area identification and isolation, respectively. While the latter has a fixed structure, the former might be either fixed or programmable, depending on designer needs (e.g. it might be automatically derived from the associated guard).

After the freezing procedure terminates the evolutionary algorithm starts (box labeled by dostrategy$_i$ in Figure 5(b)), and the base-level resumes from the "suspended" state: what is implicitly accomplished by setting no dependence between the priority of dostrategy$_i$ subnet transitions (arbitrarily assigned by the meta-

programmer) and the priority of base-level PN transitions (in practice: setting the base-level PN lowest priority equal to the priority level, assumed constant, of dostrategy$_i$ subnet). The only forced constraint is that dostrategy$_i$ submodel can *exclusively* manipulate (by means of framework's evolutionary interface) the nodes of base-level reification belonging to the pattern previously computed (this constraint is graphically expressed in Figure 5(b) by an arc between dostrategy$_i$ box and place «pattern$_i$»). As soon as the base-level PN enters a new state (marking), the newly entered base-level state is instantaneously reified into the meta-level. This reification does not involve the base-level area touched by the evolutionary strategy, which can continue operating without inconsistency. Before activating the final shift-down (which ends the strategy and actually operates the base-level evolution planned by the strategy) the temporary isolated influence area is unfrozen in a very simple way.

The described approach is more flexible than a brute-force blocking one (where the base-level is suspended for the whole duration of the strategy) while guaranteeing a sound and consistent system evolution. It better adheres to the semantics and the behavior of most real systems (think e.g. of a traffic control system), which cannot be completely suspended while their evolution is being planned.

Casually Connecting the Base-Level and the Meta-Program

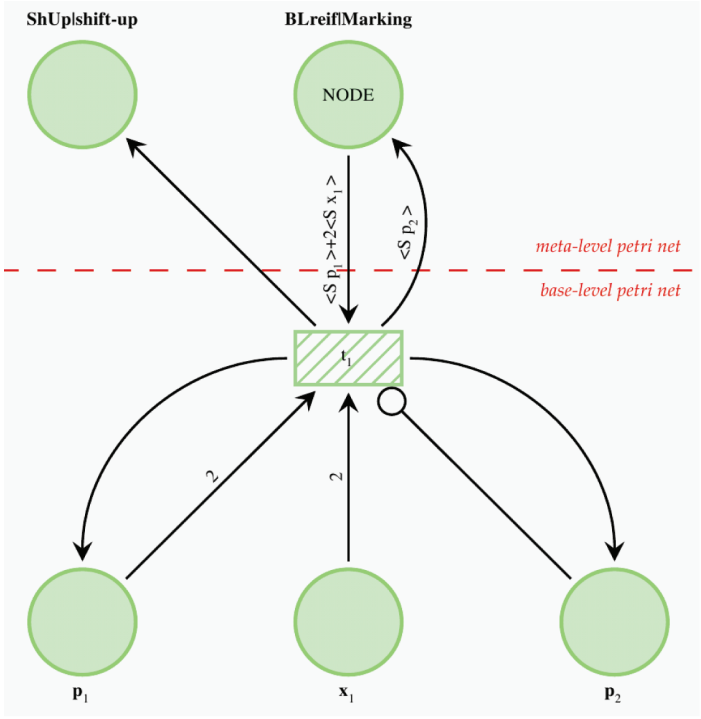
The base-level and the meta-program are (reciprocally) causally connected via the reflective framework.

Shift-up action. The shift-up action is realized for the first time at system start-up. The idea (illustrated in Figure 7) is to connect in transparent, fully automatic way the base-level PN to the evolutionary framework interface by means of colored input/output arcs drawn from any base-level PN transition to place *BLreif | Marking* of base-level reification. Any change of state at base-level PN provoked by transition firing is instantaneously reproduced on the reification, conceptually maintaining base-level unawareness about the meta-program. The firing of base-level transition t_1 results in withdrawing one and two tokens from places p_1 and x_1, respectively, and in putting one in p_2. While token consumption is emulated by a suitable input arc function ($\langle S\,p_1\rangle + 2\cdot\langle S\,x_1\rangle$), token production is emulated by an output arc function ($\langle S\,p_2\rangle$). The complete splitting of class *NODE* allows anonymous places introduced into the base-level (x_1) to be referred to by means of SWN constant functions. The occurrence of transition t_1 is signaled to the meta-program by putting one token in the uncolored boundary-place ShUp| shift-up (Figure 5(a)).

Shift-down action. The shift-down action is the only operation that cannot be directly emulated at Petri nets (WN) level, but that should be managed by the environment supporting the reflective architecture simulation. This is not surprising, rather is a consequence of the adopted choice of a traditional Petri nets paradigm to model an evolutionary architecture. The shift-down action takes place when the homonym uncolored (meta-) transition of the framework (Figure 3) is enabled. This transition has the highest priority within the whole reflective model, its occurrence replaces the current base-level PN with the Petri net described by the current reification, according to Definition 5. After a shift-down the base-level restarts from the (new) base-level initial marking, while the meta-program continues executing from the state preceding the shift-down.

Putting all together. The behavior of the whole reflective model (composed of the base-level PN, the evolutionary framework interface and the meta-program) between consecutive shift-downs can be represented using a uniform, Petri net-based approach. We are planning to extend the GreatSPN tool (Chiola, Franceschinis, Gaeta, & Ribaudo, 1995), which supports the GSPN and SWN formalisms, to be used as editing/simula-

Figure 7. Reification implemented at Petri net level



tion environment of Reflective Petri nets. For that purpose it should be integrated with a module implementing the causal-connection between base-level and meta-program.

The reflective framework, the evolutionary meta-program, and the base-level are separated sub-models, sharing three disjoint sets of boundary places: the base-level reification, the evolutionary interface, and the places holding basic command results. Their interaction is simply achieved through *superposition* of homonym places. This operation is supported by the Algebra module (Bernardi, Donatelli, & Horvàth, 2001) of GreatSPN.

Following the model composition, absolute priority levels must be set, respecting the reciprocal constraints between components earlier discussed (e.g. framework's lowest priority must be grater than meta-program's highest priority). Finally, the whole model's initial marking is set according to Definition 3 as concerns base-level reification, putting token *null* in both places Res|newP and Res|newT (Figure 4), and one uncolored token in place startMetaProgram (Figure 5(a)).

Meta-Language Basic Elements

The meta-programming language disposes of four built-in types NAT, BOOL, NODE, ArcType and the **Set** and *Cartesian product* (×) constructors. The arc (ARC: NODE × NODE × ArcType), arc with multiplicity (ArcM: ARC × NAT), and marking (Mark: NODE × NAT) types are thus introduced, this way a multi-set can be represented as a set. Place, Tran and static subclass names can be used to denote subtypes or constants (in case of singletons), and new types can be defined on-the-fly by using set operators.

Each strategy is defined in terms of basic actions, corresponding to the basic commands previously described. Their signatures are:

- newNode(Set(NODE)), newPlace(), newTran(), remNode(Set(NODE));
- flush(Set(Place))
- addArc(Set(ArcM)), remArc(Set(Arc));
- incMark(Set(Mark)), decMark(Set(Mark))
- setPrio(Set(Tran))

A particular version of repetitive command can be used. Letting E_i be a *set* (Grammar 1):

*(e_1 in E_1, ..., e_n in E_n)[«command»]

makes the instruction «command» be executed iteratively for each $e_1 \in E_1$,.., $e_n \in E_n$; at each iteration, variables e_1,.., e_n are bound to particular elements of E_1,.., E_n, respectively. If E_1 is a color (sub-)class, then we implicitly refer to its elements that belong to the base-level reification.

The meta-programmer can refer to base-level elements either explicitly, by means of constants, or implicitly, by means of *variables*.

By means of the assignments p=newPlace(), t=newTran(), it is also possible to add unspecified nodes to the base-level, afterwards referred to by variables p,t.

Base-level introspection is carried out by means of simple net-expressions allowing the meta-programmer to specify patterns, i.e., parametric base-level portions meeting some requirements on base-level's structure/marking.

The syntax for patterns and guards is shown in Grammar 1. The symbols: pre(n), post(n), inh(n), #p, card(a) denote the pre/post-sets of a base-level PN node n, the set of elements connected to n via inhibitor arcs, the current marking of place p, and the multiplicity of an arc a, respectively. They are translated into introspection commands (Figure 4). A pattern example is:

```
{p:Place|#p > #p1 and isempty
(pre(p)∩inh(p))},
```

where p1 is a constant, and p is a variable.

Below is an example of guard is (in the current version of the language quantifiers cannot be nested):

```
exists t:Tran|isempty (pre(t) ∪
inh(t)).
```

Having at our disposal a simple meta-programming language, it becomes easier specifying (even complex) parametric base-level evolutions, such as "for each marked place p belonging to the preset of t, if there is no inhibitor arc connecting p and t, add one with cardinality equal to the marking of p", which becomes:

*(p in pre(t)) [#p>0 and card(<p,t,h>)==0 → addArc(<p,t,h,#p>)]

The code of the freezing algorithm acting on a precomputed influence area (box isolate(«$pattern_i$»)) in Figure 5(b)), which is one of the fixed parts of the meta-program, is given in Figure 8. all base-level transitions that belong to the pattern, or that can change its local marking (state), are temporarily prevented from firing by adding a new (marked) place to the base-level reification, to which pattern transitions are connected via inhibitor arcs. A shift-down action is then activated to freeze the base-level PN. Unfreezing is simply achieved by removing the artificially introduced inhibitor place at the end of the evolutionary strategy (Figure 5(b)).

```
[
isempty(pattern) → skip
□
not(isempty(pattern)) →
pattern* = {};
isolating_pattern = newPlace();
incMark(<isolating_pattern,1>);
*(p in Pattern ∩ Place)
[true → pattern* ∪ = pre(p) ∪ post(p)];
*(t in pattern* ∩ Tran)
```

Figure 8. CSP Code for the Isolating-Pattern Subnet (Language's Keywords are in Bold)

```
[
 isempty(pattern) → skip
  □
 not(isempty(pattern)) →
   pattern* = {};
   isolating_pattern = newPlace();
   incMark(<isolating_pattern,1>);
   *(p in Pattern □ Place)
     [true → pattern* □ = pre(p) □ post(p)];
   *(t in pattern* □ Tran)
     [true →newArc(<isolating_pattern,t,h,1>)];
   shiftDown;
]
```

[**true** →**newArc**(<isolating_pattern,t,h,1>)];
shiftDown;
]

A MARKOV-PROCESS FOR REFLECTIVE PETRI NETS

The adoption of GSPN (Ajmone Marsan, Conte, & Balbo, 1984) and SWN (Chiola, Dutheillet, Franceschinis, & Haddad, 1993) for the base- and meta- levels of the reflective layout, respectively, has revealed a convenient choice for two reasons: first, the timed semantics of Reflective Petri nets is in large part inherited from GSPN (SWN); secondly, the symbolic marking representation the SWN formalism is provided with can be exploited to efficiently handle the intriguing question related to how identifying equivalences during a Reflective Petri net model evolution.

On the light of the connection set between base- and meta- levels, the behavior of a Reflective Petri net model between any meta-level activation and the consequent shift-down is fully described in terms of a SWN model, the meta-level PN, including (better, suitably connected to) an uncolored part (the base-level PN). This model will be hereafter denoted *base-meta PN*. Hence, we can naturally set the following notion of state for Reflective Petri nets:

Definition 6 (state). *A state of a Reflective Petri net is a marking* $\mathbf{M}_i$ *of the base-meta PN obtained by suitably composing the base-level PN (a GSPN) and the meta-level PN (a SWN).*

Then, letting $t \neq$ shiftdown be any transition (color instance) enabled in $\mathbf{M}_i$, according to the SWN (GSPN) firing rule, and $\mathbf{M}_j$ be the marking reached upon its firing, we have the labeled state-transition

$$\mathbf{M}_i \xrightarrow{\lambda(t)} \mathbf{M}_j,$$

where $\lambda(t)$ denotes a weight, or an exponential rate, associated with t, depending on whether t is timed or immediate.

There is nothing to do but consider the case where $\mathbf{M}_f$ is a vanishing marking enabling the pseudo-transition *shift-down*: then,

$$\mathbf{M}_f \xrightarrow{w=1} \mathbf{M}'_0,$$

$\mathbf{M}'_0$ being the marking of the base-meta PN obtained by first replacing the (current) base-level PN with the GSPN isomorphic to the reification marking (once it has been suitably connected to

Table 2.

Grammar **1 BNF for language expressions.**		
Element	::=	NODE | Arc†
NODE	::=	«*variable*» | «*constant*» | **singleton (** NodeSet **)**
Arc	::=	< NODE , NODE , «*arc_type*» >
Expression	::=	«*digit*» | BasicExpr
BasicExpr	::=	# «*place*»‡ | **card(** Set **)** | **card (** Arc **)** | **prio (** «*transition*» **)**
Predicate	::=	BasicExpr RelOp Expression | **kind (** Arc **)** EqOp «*arc_type*» | NODE InExpr | NODE **is connected to** NODE | **isempty (** Set **)**
RelOp	::=	< | > | =
EqOp	::=	=\= | =
Set	::=	{ } | { ArcList } | NodeSet | «*static_subclass*» | «*color_class*» | Element | Set SetOp Set
SetOp	::=	∩ | ∪ | \
ArcList	::=	Arc | ArcList **,** Arc
NodeSet	::=	{ } | { NodeList } | Pattern | AlgOp **(** NodeSet **)** | NODE
NodeList	::=	NODE | NodeList **,** NODE
AlgOp	::=	**pre** | **post** | **inh**
Pattern	::=	{ «*variable*» InExpr **|** Guard }
Guard	::=	Predicate | LogOp «*variable*» InExpr Predicate | **not (** Guard **)**
InExpr	::=	∈ | **in** «*place*» | **in** NodeSet
LogOp	::=	**exists** | **foreach**
BoolOp	::=	**and** | **or**

† Terminals are in bold font, non-terminals are in normal font. ‡ Terms in «» represent elements whose meaning can be inferred from the model.

the meta-level PN), then firing *shift-down* as it were a normal immediate transition.

Using the same technique for eliminating vanishing states as it is employed in the *reduced* reachability graph algorithm (Ajmone Marsan, Conte, & Balbo, 1984), it is possible to build a CTMC for the Reflective Petri net model.

Recognizing Equivalent Evolutions

The state-transition graph semantics just introduced precisely defines the (timed) behavior of a Reflective Petri net model, but suffers from two evident drawbacks. First, it is highly inefficient: the state description is exceedingly redundant, comprising a large part concerning the meta-level PN, which is unnecessary to describe the evolving system. The second concern is even more critical, and indirectly affects efficiency: there is no way of recognizing whether the modeled system, during its dynamics/evolution, reaches equivalent states. The ability of deciding about a system's state-transition graph finiteness and strongly-connectedness, of course strictly related to the ability of recognizing equivalent states, is in fact mandatory for performance analysis: we know that the most important sufficient condition for a finite CTMC to have stationary solution (steady-state) is to include one maximal strongly connected component.

More generally, most techniques based on state-space inspection rely on the ability above.

Recognizing equivalent evolutions is a tricky question. For example, it may happen that (apparently) different strategies cause in truth equivalent transformations to the base-level PN (the evolving system), which cannot be identified by Definition 6. Yet, the combined effect of different sequences of evolutionary strategies might produce the same effects. Even more likely, the internal dynamics of the evolving system might lead to reach equivalent configurations. The above question, which falls into a graph isomorphism sub-problem, as well as the global efficiency of the approach, are tackled by resorting to the peculiar characteristic of SWN: the symbolic marking notion (Chiola, Dutheillet, Franceschinis, & Haddad, 1997).

For that purpose, we refer to the following static partition of class *NODE*:

$$NODE = \underbrace{\overbrace{p_1 \cup \ldots p_k}^{Named_p} \cup Unnamed_p}_{Place} \cup \underbrace{\overbrace{t_1 \cup \ldots t_n}^{Named_t} \cup Unnamed_t}_{Tran}.$$

Symbols p_i, t_j denote singleton static subclasses. Conversely, $Unnamed_p$ and $Unnamed_t$ are static subclasses collecting all anonymous (i.e., indistinguishable) places/transitions. Behind there is a simple intuition: while some ("named") nodes, for the particular role they play, preserve the identity during base-level evolution, and may be explicitly referred to during base-level manipulation, others ("unnamed") are indistinguishable from one another. In other words any pair of "unnamed" places (transitions) might be freely exchanged on the base-level PN, without altering the model's semantics. There are two extreme cases: $Named_p$ ($Named_t$) = ∅ and, opposite, $Unnamed_p$ ($Unnamed_t$) = ∅. The former meaning all places/transitions can be permuted, the latter instead all nodes are distinct.

It is remarkable that the static partition of class *NODE* actually used for the base-meta PN is different from the previous one, given that any places of base-level PN must be explicitly referred to when connecting the base-level PN to the meta-level PN (Figure 7).

The technique we use to recognize equivalent base-level evolutions relies on the base-level reification and the adoption of a symbolic state representation for the base-meta PN that, we recall, results from composing in transparent way the base-level PN and the meta-level PN.

We only have to set as initial state of the Reflective Petri net model a symbolic marking ($\hat{\mathbf{M}}_0$) of the base-meta PN, instead of an ordinary one: any dynamic subclass of $Unnamed_p$ ($Unnamed_t$) will represent an arbitrary "unnamed" place (transition) of the base-level PN.

Because of the simultaneous update mechanism of the reification, and the consequent one-to-one correspondence along the time between the current base-level PN and the reification at the meta-level, we can state the following:

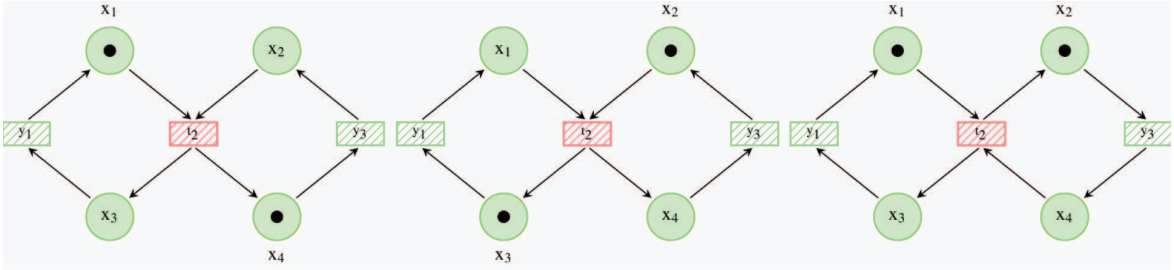
Definition 7 (equivalence relation) *Let* $\hat{\mathbf{M}}_i$, $\hat{\mathbf{M}}_j$ be two symbolic states of the Reflective Petri net model. $\hat{\mathbf{M}}_i \equiv \hat{\mathbf{M}}_j$ if and only if their restrictions on the reification set of places have the same canonical representative.

Lemma 1. *Let* $\hat{\mathbf{M}}_i \equiv \hat{\mathbf{M}}_j$. Then the base-level PNs at states $\hat{\mathbf{M}}_i$ and $\hat{\mathbf{M}}_j$ are isomorphic.

Consider the very simple example in Figure 9, which depicts three base-level PN configurations, at different time instants. The hypothesis is that while symbol t_2 denotes a ''named'' transition, symbols x_i and y_j denote ''unnamed'' places and transitions, respectively. Since there are no inhibitor arcs we assume that arcs are reified as tokens (2-tuples) belonging to *NODE* × *NODE*. We assume that all transitions have the same priority level, so we disregard the reification of priorities.

We can observe that the Petri nets on the left and on the middle are nearly the same, but for their current marking: we can imagine that they represent a possible (internal) dynamics of the base-level PN. Conversely, we might think of the right-most Petri net as an (apparent) evolution of the base-level PN on the left, in which transition y_2 has been replaced by the (new) transition y_3, new connections are set, and a new marking is defined.

Figure 9. Equivalent Base-Level Petri Net Evolutions



Nevertheless, the three base-level configurations are equivalent, according to Definition 7. It is sufficient to take a look at their respective reifications, which are encoded as symbolic markings (multisets are expressed as formal sums): consider for instance the base-level PN on the left and on the middle of Figure 9, whose reification are:

$$\hat{\mathbf{M}}(\text{BLreif} \mid \text{Nodes}) = y_1 + y_2 + t_2 + x_1 + x_2 + x_3 + x_4,$$

$$\hat{\mathbf{M}}(\text{BLreif} \mid \text{Marking}) = x_1 + x_4,$$

$$\hat{\mathbf{M}}(\text{BLreif} \mid \text{Arcs}) = \langle x_1, t_2\rangle + \langle t_2, x_3\rangle + \langle x_3, y_1\rangle + \langle y_1, x_1\rangle + \langle x_2, t_2\rangle +$$

$$\langle t_2, x_4\rangle + \langle x_4, y_2\rangle + \langle y_2, x_2\rangle$$

and

$$\hat{\mathbf{M}}'(\text{BLreif} \mid \text{Nodes}) = y_1 + y_2 + t_2 + x_1 + x_2 + x_3 + x_4,$$

$$\hat{\mathbf{M}}'(\text{BLreif} \mid \text{Marking}) = x_3 + x_2,$$

$$\hat{\mathbf{M}}'(\text{BLreif} \mid \text{Arcs}) = \langle x_1, t_2\rangle + \langle t_2, x_3\rangle + \langle x_3, y_1\rangle + \langle y_1, x_1\rangle +$$

$$\langle x_2, t_2\rangle + \langle t_2, x_4\rangle + \langle x_4, y_2\rangle + \langle y_2, x_2\rangle$$

respectively. They can be obtained from one another by the following permutation of "unnamed" places and transitions (we denote by $a \leftrightarrow b$ the bidirectional mapping: $a \rightarrow b, b \rightarrow a$):

$$\{x_1 \leftrightarrow x_2, x_3 \leftrightarrow x_4, y_1 \leftrightarrow y_2\},$$

thus, they are equivalent.

With similar arguments, the left-most and the right-most Petri nets of Figure 9 are shown to be equivalent. The left-most Petri net's reification is:

$$\hat{\mathbf{M}}''(\text{BLreif} \mid \text{Nodes}) = y_1 + y_3 + t_2 + x_1 + x_2 + x_3 + x_4,$$

$$\hat{\mathbf{M}}''(\text{BLreif} \mid \text{Marking}) = x_1 + x_2,$$

$$\hat{\mathbf{M}}''(\text{BLreif} \mid \text{Arcs}) = \langle x_1, t_2\rangle + \langle t_2, x_3\rangle + \langle x_3, y_1\rangle + \langle y_1, x_1\rangle +$$

$$\langle x_2, y_3\rangle + \langle y_3, x_4\rangle + \langle x_4, t_2\rangle + \langle t_2, x_2\rangle$$

$\hat{\mathbf{M}}$ and $\hat{\mathbf{M}}''$ can be obtained from one another through the following permutation:

$$\{x_2 \leftrightarrow x_4, y_3 \leftrightarrow y_2\},$$

The canonical representative for these equivalent base-level PN's reifications (i.e., states of the Reflective Petri net model), computed according to the corresponding SWN algorithm, turns out to be $\hat{\mathbf{M}}$.

RELATED WORKS

Although many other models of concurrent and distributed systems have been developed, Petri Nets are still considered a central model for concurrent systems with respect to both the theory and the applications due to the natural way they

allow to represent reasoning on concurrent active objects which share resources and their changing states. Despite their modeling power (Petri nets with inhibitor arcs are Turing-equivalent) however, classical Petri nets are often considered unsuiTable to model real systems. For that reason, several high-level Petri nets paradigms (Colored Petri nets, Predicate/Transition Nets, Algebraic Petri nets) have been proposed in the literature (Jensen & Rozenberg, 1991) over the last two decades to provide modelers with a more flexible and parametric formalism able to exploit the symmetric structure of most artificial discrete-event systems.

Modern information systems are more and more characterized by a dynamic/reconfigurable (distributed) topology and they are often conceived as self-evolving structures, able to adapt their behavior and their functionality to environmental changes and new user needs. Evolutionary design is now a diffuse practice, and there is a growing demand for modeling/simulation tools that can better support the design phase. Both Petri nets and HLPN are characterized by a fixed structure (topology), so many research efforts have been devoted, especially in the last two decades, in trying to extend Petri nets with dynamical features. Follows a non-exhaustive list of proposals appeared in the literature.

In Valk, 1978, the author is proposing his pioneering work, *self-modifying* nets. Valk's self-modifying nets introduce dynamism via self modification. More precisely the flow relation between a place and a transition is a linear function of the place marking. Techniques of linear algebra used in the study of the structural properties of Petri nets can be adapted in this extended framework. Only simple evolution patterns can be represented using this formalism. Another major contribution of Valk is the so-called *nets-within-nets* paradigm (Valk, 1998), a multi-layer approach, where tokens flowing through a net are in turn nets. In his work, Valk takes an object as a token in a unary elementary Petri net system, whereas the object itself is an elementary net system. So, an object can migrate across a net system. This bears some resemblance with logical agent mobility. Even if in the original Valk's proposal no dynamic changes are possible, many dynamic architectures introduced afterward (including in some sense also the approach proposed in this chapter) rely upon his paradigm.

Some quite recent proposals have extended Valk's original ideas. Badouel & Darondeau, 1997 introduces a subclass of self-modifying nets. The considered nets appear as stratified sums of ordinary nets and they arise as a counterpart to cascade products of automata via the duality between automata and nets. Nets in this class, called *stratified nets*, cannot exhibit circular dependences between places: inscription on flow arcs attached to a given place depends at most on the content of places in the lower layers. As an attempt to add modeling flexibility, Badouel & Oliver, 1998 defines a class of high-level Petri nets, called *reconfigurable nets*, that can dynamically modify their own structure by rewriting some of their components. Boundedness of a reconfigurable net can be decided by calculating its covering tree. Moreover such a net can be simulated by a self-modifying Petri net. The class of reconfigurable nets thus provides a subclass of self-modifying Petri nets for which boundedness can be decided.

Modeling mobility, both physical and logical, is another active subject of ongoing research. Mobile and dynamic Petri nets (Asperti & Busi, 1996) integrate Petri nets with RCHAM (Reflective Chemical Abstract Machine) based process algebra. In dynamic nets tokens are names for places, an input token of a transition can be used in its postset to specify a destination, and moreover the creation of new nets during the firing of a transition is also possible. Mobile Petri nets handle mobility expressing the configuration changing of communication channels among processes.

Tokens in Petri nets, even in self-modifying, mobile/dynamic and reconfigurable nets, are passive, whereas agents are active. To bridge the gap

between tokens and agents, or active objects, many authors have proposed variations on the theme of *nets-within-nets*. In Farwer & Moldt, 2005, objects are studied as higher-level net tokens having an individual dynamical behavior. Object nets behave like tokens, i.e., they are lying in places and are moved by transitions. In contrast to ordinary tokens, however, they may change their state. By this approach an interesting two-level system modeling technique is introduced. Xu, Yin, Deng, & Ding, 2003 proposes a two-layers approach. From the perspective of system's architecture, it presents an approach to modeling logical agent mobility by using Predicate Transition nets as formal basis for the dynamic framework. Reference nets proposed in Kummer, 1998 are another formalism based on Valk's work. Reference nets are a special high level Petri net formalism that provide dynamic creation of net instances, references to other reference nets as tokens, and communication via synchronous channels (Java is used as inscription language).

Some recent proposals have some similarity with the work we are presenting in this chapter or, at least, are inspired by similar aims. In Cabac et al., 2005 the authors present the basic concepts for a dynamic architecture modeling (using nets-within-nets) that allows active elements to be nested in arbitrary and dynamically changeable hierarchies and enables the design of systems at different levels of abstractions by using refinements of net models. The conceptual modeling of such architecture is applied to specify a software system that is divided into a plug-in management system and plug-ins that provide functionality to the users. By combining plug-ins, the system can be dynamically adapted to the users needs. In Hoffmann et al., 2005 the authors introduce the new paradigm of *nets and rules as tokens*, where in addition to nets as tokens also rules as tokens are considered. The rules can be used to change the net structure and behavior. This leads to the new concept of high-level net and rule systems, which allows to integrate the token game with rule-based transformations of P/T-systems. The new concept is based on algebraic nets and graph transformation systems. Finally, in Odersky, 2000 the author introduces functional nets, which combine key ideas of functional programming and Petri nets to yield a simple and general programming notation. They have their theoretical foundation in join calculus. Over the last decade an operational view of program execution based on rewriting has become widespread. In this view, a program is seen as a term in some calculus, and program execution is modeled by stepwise rewriting of the term according to the rules of the calculus.

All these formalisms, however, set up new hybrid (high-level) Petri net-based paradigms. While the expressive power has increased, the cognitive simplicity, which is the most important advantage of Petri nets, has decreased as well. In Badouel, 1998 the authors argued that the intricacy of these models leaves little hope to obtain significant mathematical results and/or automated verification tools in a close future. The approach we are presenting differs from the previous ones mainly because it achieves a satisfactory compromise between expressive power and analysis capability, through a quite rigorous application of classical reflection concepts in a consolidated (high-level) Petri net framework.

CONCLUSION AND FUTURE WORK

Most discrete-event systems are subject to evolution, and need to be updated or extended with new characteristics during lifecycle. Covering the evolutionary aspects of systems since the design phase has been widely recognized as a crucial challenge. A good evolution has to pass through the evolution of the design information of the system itself. Petri nets are a central formalism for the modeling of discrete-event systems. Unfortunately classical Petri nets have a static structure, so Petri net modelers are forced to hard-code all the foreseeable evolutions of a system at the design level.

This common practice not only requires modeling expertise, it also makes system's design be polluted by lot of details that do not regard the (current) system functionality, and affect the consolidated Petri nets analysis techniques.

We have faced the problem through the definition of a Petri net-based reflective architecture, called Reflective Petri Nets, structured in two logical levels: the base-level, specifying the evolving system, and the evolutionary meta-program (the meta-level). The meta-program is in charge of observing in transparent way, then (if necessary) transforming, the base-level PN. With this approach the model of the system and the model of the evolution are kept separated, granting, therefore, the opportunity of analyzing the model without useless details. The evolutionary aspects are orthogonal to the functional aspects of the system.

In this chapter we have introduced Reflective Petri nets, and we propose an effective timed state-transition semantics (in terms of a Markov process) as a first step toward the implementation of a (performance-oriented) discrete-event simulation engine for Reflective Petri nets. Ongoing research is in different directions. We are planning to extend the GreatSPN tool to directly support Reflective Petri nets, both in the editing and in the analysis/simulation steps. We are investigating other possible semantic characterizations (in terms of different stochastic processes), on the perspective of improving the analysis capability. We are currently using two different formalisms for the base- and meta- levels (ordinary and colored stochastic Petri nets). It might be convenient to adopt the same formalism for both levels, what would give origin to the reflective tower allowing the designer to model also the possible evolution of the evolutionary strategies.

REFERENCES

Asperti, A., & Busi, N. (1996, May). *Mobile Petri Nets* (Tech. Rep. No. UBLCS-96-10). Bologna, Italy: Università degli Studi di Bologna.

Badouel, E., & Darondeau, P. (1997, September). Stratified Petri Nets. In B. S. Chlebus & L. Czaja (Eds.), *Proceedings of the 11th International Symposium on Fundamentals of Computation Theory (FCT'97)* (p. 117-128). Kraków, Poland: Springer.

Badouel, E., & Oliver, J. (1998, January). *Reconfigurable Nets, a Class of High Level Petri Nets Supporting Dynamic Changes within Workflow Systems* (IRISA Research Report No. PI-1163). IRISA.

Bernardi, S., Donatelli, S., & Horvàth, A. (2001, September). Implementing Compositionality for Stochastic Petri Nets. *Journal of Software Tools for Technology Transfer*, *3*(4), 417–430.

Best, E. (1986, September). COSY: Its Relation to Nets and CSP. In W. Brauer, W. Reisig, & G. Rozenberg (Eds.), *Petri Nets: Central Models and Their Properties, Advances in Petri Nets (Part II)* (p. 416-440). Bad Honnef, Germany: Springer.

Cabac, L., Duvignau, M., Moldt, D., & Rölke, H. (2005, June). Modeling Dynamic Architectures Using Nets-Within-Nets. In G. Ciardo & P. Darondeau (Eds.), *Proceedings of the 26th International Conference on Applications and Theory of Petri Nets (ICATPN 2005)* (p. 148-167). Miami, FL: Springer.

Capra, L., & Cazzola, W. (2007, December). Self-Evolving Petri Nets. *Journal of Universal Computer Science*, *13*(13), 2002–2034.

Capra, L., & Cazzola, W. (2009). Trying out Reflective Petri Nets on a Dynamic Workflow Case. In E. M. O. Abu-Atieh (Ed.), *Handbook of Research on Discrete Event Simulation Environments Technologies and Applications*. Hershey, PA: IGI Global.

Capra, L., De Pierro, M., & Franceschinis, G. (2005, June). A High Level Language for Structural Relations in Well-Formed Nets. In G. Ciardo & P. Darondeau (Eds.), *Proceeding of the 26th international conference on application and theory of Petri nets* (p. 168-187). Miami, FL: Springer.

Cazzola, W. (1998, July 20th-24th). Evaluation of Object-Oriented Reflective Models. In *Proceedings of ecoop workshop on reflective object-oriented programming and systems (ewroops'98)*. Brussels, Belgium.

Cazzola, W., Ghoneim, A., & Saake, G. (2004, July). Software Evolution through Dynamic Adaptation of Its OO Design. In H.-D. Ehrich, J.-J. Meyer, & M. D. Ryan (Eds.), *Objects, Agents and Features: Structuring Mechanisms for Contemporary Software* (pp. 69-84). Heidelberg, Germany: Springer-Verlag.

Chiola, G., Dutheillet, C., Franceschinis, G., & Haddad, S. (1990, June). On Well-Formed Coloured Nets and Their Symbolic Reachability Graph. In *Proceedings of the 11th international conference on application and theory of Petri nets*, (p. 387-410). Paris, France.

Chiola, G., Dutheillet, C., Franceschinis, G., & Haddad, S. (1993, November). Stochastic Well-Formed Coloured Nets for Symmetric Modeling Applications. *IEEE Transactions on Computers*, *42*(11), 1343–1360. doi:10.1109/12.247838

Chiola, G., Franceschinis, G., Gaeta, R., & Ribaudo, M. (1995, November). GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Performance Evaluation*, *24*(1-2), 47–68. doi:10.1016/0166-5316(95)00008-L

Farwer, B., & Moldt, D. (Eds.). (2005, August). *Object Petri Nets, Process, and Object Calculi*. Hamburg, Germany: Universität Hamburg, Fachbereich Informatik.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Upper Saddle River, NJ: Prentice Hall.

Hoffmann, K., Ehrig, H., & Mossakowski, T. (2005, June). High-Level Nets with Nets and Rules as Tokens. In G. Ciardo & P. Darondeau (Eds.), *Proceedings of the 26th International Conference on Applications and Theory of Petri Nets* (p. 268-288). Miami, FL: Springer

Hürsch, W., & Videira Lopes, C. (1995, February). *Separation of Concerns* (Tech. Rep. No. NUCCS-95-03). Northeastern University, Boston.

Jensen, K., & Rozenberg, G. (Eds.). (1991). *High-Level Petri Nets: Theory and Applications*. Berlin: Springer-Verlag.

Kavi, K. M., Sheldon, F. T., Shirazi, B., & Hurson, A. R. (1995, January). Reliability Analysis of CSP Specifications Using Petri Nets and Markov Processes. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences (HICSS-28)* (p. 516-524). Kihei, Maui, HI: IEEE Computer Society.

Kummer, O. (1998, October). Simulating Synchronous Channels and Net Instances. In J. Desel, P. Kemper, E. Kindler, & A. Oberweis (Eds.), *Proceedings of the Workshop Algorithmen und Werkzeuge für Petrinetze* (Vol. 694, pp. 73-78). Dortmund, Germany: Universität Dortmund, Fachbereich Informatik.

Maes, P. (1987, October). Concepts and Experiments in Computational Reflection. In N. K. Meyrowitz (Ed.), *Proceedings of the 2nd conference on object-oriented programming systems, languages, and applications (OOPSLA'87)* (Vol. 22, p. 147-156), Orlando, FL.

Odersky, M. (2000, March). Functional Nets. In G. Smolka (Ed.), *Proceedings of the 9th European Symposium on Programming (ESOP 2000)* (p. 1-25). Berlin, Germany: Springer.

Valk, R. (1978, July). Self-Modifying Nets, a Natural Extension of Petri Nets. In G. Ausiello & C. Böhm (Eds.), *Proceedings of the Fifth Colloquium on Automata, Languages and Programming (ICALP'78),* (p. 464-476). Udine, Italy: Springer.

Valk, R. (1998, June). Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In J. Desel & M. Silva (Eds.), *Proceedings of the 19th International Conference on Applications and Theory of Petri Nets (ICATPN 1998)* (p. 1-25). Lisbon, Portugal: Springer.

Xu, D., Yin, J., Deng, Y., & Ding, J. (2003, January). A Formal Architectural Model for Logical Agent Mobility. *IEEE Transactions on Software Engineering*, *29*(1), 31–45. doi:10.1109/TSE.2003.1166587

KEY TERMS AND DEFINITIONS

Evolution: Attitude of systems to change layout/functionality.

Dynamic Systems: Discrete-event systems subject to evolution.

Petri Nets: Graphical formalism for discrete-event systems.

Reflection: Activity performed by an agent when doing computations about itself.

Base-Level: Logical level of a reflective model representing the system prone to evolve.

Meta-Level: Logical level of a reflective model representing the evolutionary strategy.

State-Transition Graph: Graph describing the behavior of a system in terms of states and transitions between them.

ENDNOTES

1 Labels taking the form place_name | postfix denote *boundary-places*

2 Recall that: i) CSP is based on *guarded-commands*; ii) structured commands are included between square brackets; and iii) symbols ?, *, and □ denote input, *repetition* and *alternative* commands, respectively.