Journal of Universal Computer Science, vol. 13, no. 13 (2007), 2002-2034
submitted: 30/9/06, accepted: 15/11/07, appeared: 28/12/07 © J.UCS

Self-Evolving Petri Nets

Lorenzo Capra
(Department of Informatics and Communication
Università degli Studi di Milano, Italy
capra@dico.unimi.it)

Walter Cazzola
(Department of Informatics and Communication
Università degli Studi di Milano, Italy
cazzola@dico.unimi.it)

Abstract: Nowadays, software evolution is a very hot topic. It is particularly complex when it regards critical and nonstopping systems. Usually, these situations are tackled by hard-coding all the foreseeable evolutions in the application design and code.

Neglecting the obvious difficulties in pursuing this approach, we also get the application code and design polluted with details that do not regard the current system functionality, and that hamper design analysis, code reuse and application maintenance in general. Petri Nets (PN), as a formalism for modeling and designing distributed/concurrent software systems, are not exempt from this issue.

The goal of this work is to propose a PN based reflective framework that lets everyone model a system able to evolve, keeping separated functional aspects from evolutionary ones and applying evolution to the model only if necessary. Such an approach tries to keep system's model as simple as possible, preserving (and exploiting) ability of formally verifying system properties typical of PN, granting at the same time adaptability.

Key Words: Petri Nets, Reflection, Software Evolution.
Category: D.2, D.2.2, D.2.7, D.1, D.1.5

1 Introduction

Software evolution is becoming a very hot topic. Many applications need to be updated or extended with new features during lifecycle. Software evolution can imply complete system redesign, development of new features and their integration in running systems. Evolution often takes place by foreseeing how software could evolve at design-time (before it really needs to evolve) or by directly patching software without analyzing situation and planning evolution itself.

A good evolution is carried out through evolution of system design information, and then through propagating these changes to implementation. This approach should be the most natural and intuitive to use (because it adopts the same mechanisms adopted during development phase) and it should produce the best results (because each evolutionary step is planned and documented before its operation).

At the moment software evolution, especially when related to critical and non-stopping systems, is emulated by directly enriching original design information (and consequently code) with aspects concerning possible evolutions. This approach has several drawbacks:

- all possible evolutions are not always foreseeable;
- design information 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 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;
- code and model pollution hinders application maintenance and reduces possibility of reuse.

PN, when used to model systems that could evolve, suffer from these problems as well. At present, software evolution through evolutionary design is not supported by traditional PN classes. Normally it is achieved by merging the basic model of a software system with information on the foreseeable evolutions of the system itself. A similar approach pollutes the model with details not pertinent to the current structure of the system. Pollution not only increases complexity of formal models but hinders ability of existing tools of analyzing system properties without considering all possible branches of evolution.

System evolution is an aspect orthogonal to (current) system behavior that crosscuts both application code and design; hence it could be subject to separation of concerns [Hürsch and Videira Lopes, 1995]. Separating evolution from the rest of a system is worthwhile, because evolution is made independent of the evolving system and the abovementioned problems overcame. Separation of concerns could be applied to a PN-based modeling approach as well. Design information (in our case, a PN 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 application structure and behavior. Reflection [Maes, 1987] is one of the mechanisms that easily permits to separate this kind 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-levels 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*, that is kept *causally connected* to the original level. For details look at [Cazzola, 1998].

Similarly to what we have done in [Cazzola et al., 2004], the meta-level can be programmed to evolve the base-level structure and behavior when necessary, without polluting it with extra code. In this work we apply the same idea to PN domain. We propose a reflective framework that separates the PN describing a system from the PN that describes how this system evolves when some events occur. We here extend an early version [Capra and Cazzola, 2005], by tuning the causal connection between base-level and meta-level in order to enhance parallelism between base- and meta- computations. With respect to several proposals recently appeared with similar goals [Cabac et al., 2005, Hoffmann et al., 2005], our approach does not define a new PN paradigm, rather it sets the basis of an evolutionary reflective framework relying upon consolidated classes of PNs. That gives the possibility of using existing tools and analysis techniques in a fully orthogonal fashion. The framework can be easily integrated to the GreatSPN tool [Chiola et al., 1995], allowing software designers to analyze the current system configuration and to simulate its evolution.

The rest of the paper is structured as follows: in section 2 we briefly present the adopted PN formalisms; in section 3 we give an overview of the whole reflective framework introducing the adopted terminology, then we present the (high-level) Petri net realizing the causal connection among the other components of the system; in section 4 we show our approach in action; finally in section 5 and in section 6 we present some related work and draw our conclusions and perspectives.

2 Well-formed Nets

Colored PNs [Jensen and Rozenberg, 1991] (CPN) are a major extension of PN belonging to the High-Level PN category. This work relies upon Well-formed Nets (WN) [Chiola et al., 1990], a CPN flavor retaining expressive power, characterized by a structured syntax. This section introduces WN semi-formally, by an example. Figure 1 shows the portion of the evolutionary framework (figure 3) that removes a given node from the base-level PN modeling the system (encoded as WN marking). The removal of a node has the side-effect of provoking the removal of any arc connected to the node. 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 PNs.

A WN is a tuple $(T, P, \{C_1, \dots, C_n\}, \mathcal{C}, W^+, W^-, H, \Phi, \Pi, \mathbf{M}_0)$ where P is the finite set of *places*, T is the finite set of *transitions*. With respect to ordinary PN, places may contain "colored" tokens of different identity. $C_1, \dots, C_n$ 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 subclasses, denoted $C_i = \bigcup_j C_{i,j}$. For instance class C_1 is partitioned into $places \cup trans$. For the sake of modeling, a multi-level partitioning will be adopted, where for instance subclass $places$ might be in turn refined in $named \cup unnamed$.

$\mathcal{C}$ 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 that one talks about instances of a colored transition. In figure 1 $\mathcal{C}(t)$, $t \neq$ `delAFromToN`: C_1; $\mathcal{C}($`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 marking $\mathbf{M}$ maps each place p to a multiset $Bag(\mathcal{C}(p))$. $\mathbf{M}_0$ defines the initial marking.

W^-,W^+ and H assign each pair (t, p) an (input, output and inhibitor, respectively) arc function $\mathcal{C}(t) \to Bag(\mathcal{C}(p))$. Any arc function is a (linear combination of) *function-tuple(s)* $\langle f_1, \dots, f_n\rangle$, tuple components are called *class-functions*. Each f_i is a class-*j* function, $\mathcal{C}(t) \to Bag(C_j)$, C_j being the color class on *i*-th position in $\mathcal{C}(p)$. Letting $F : \langle f_1, \dots, f_n\rangle$ and $t_c : \langle c_1, \dots, c_m\rangle \in \mathcal{C}(t)$, then $F(t_c) = f_1(t_c) \times \dots f_n(t_c)$, where $\times$ denotes multi-set Cartesian product. Each f_i is expressed in terms of *elementary* functions: the only ones appearing in paper's WN models 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 `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 $[g] : \mathcal{C}(t) \to \{true, false\}$ 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 instance t_c *has concession* in $\mathbf{M}$ iff (i) $W^-(t, p)(t_c) \leq \mathbf{M}(p)$, (ii) $H(t, p)(t_c) > \mathbf{M}(p)$, for each place p, and (iii) $\Phi(t)(t_c) =$ true ($>, \leq, +, -$ are implicitly extended to multisets). Π assigns a priority level to each transition. t_c is *enabled* in $\mathbf{M}$ if it has concession, and no higher priority transition instances have concession in $\mathbf{M}$. It can fire, leading to $\mathbf{M}'$:

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

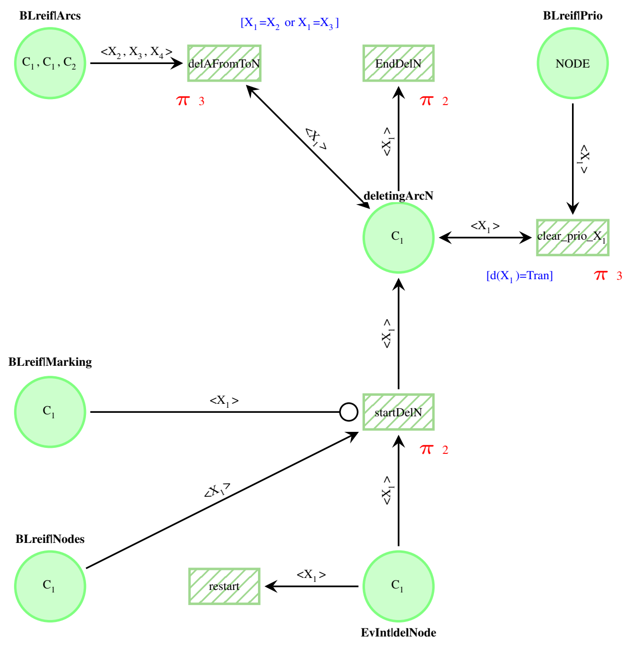


Figure 1: a Well-formed Net

Restart transitions are used in our models (again for convenience, we might always trace it back to the standard WN definition), denoted by prefix *rest*. While the enabling rule doesn't change, their firing makes a WN model go back to the initial marking.

A reachability-graph is built starting from $\mathbf{M}_0$, in stochastic WN (SWN) [Chiola et al., 1993] it is associated to a Markovian process. The particular syntax of WN arc functions allows system symmetries to be implicitly embedded into (S)WN models. This way efficient techniques can be applied, e.g. to build a compact Symbolic Reachability Graph (SRG) (and a *lumped* CTMC), or to run a symbolic discrete-event simulation.

The *base-level* class of nets (section 3) correspond to the unfolded version of WN, that is, P/T nets with priorities and inhibitor arcs. One such net is formally a tuple $(T, P, W^+, W^-, H, \Pi, \mathbf{m}_0)$, where W^+, W^-, H are functions associating to each pair (t, p) a weight in $\mathbb{N}$. Analogously, a marking $\mathbf{m}$ is a mapping $P \rightarrow \mathbb{N}$. The definitions of concession, enabling, firing given before are still valid (guards have disappeared), but for replacing $F(t, p)(t_c)$ by $F(t, p)$.

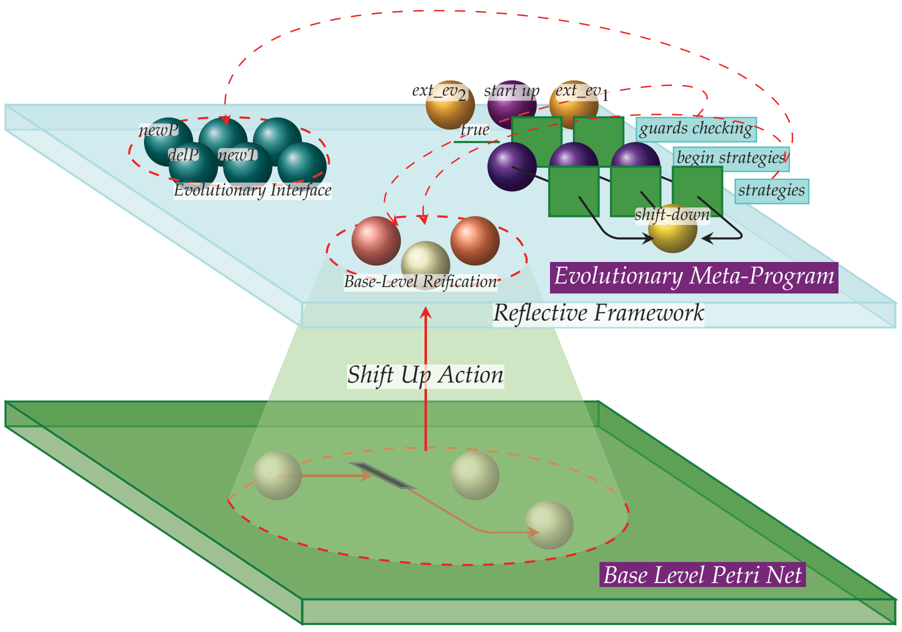


Figure 2: A snapshot of the whole reflective system: the base-level is the one prone to be adapted, the meta-level drives the adaptation.

3 A Reflective Petri Net Model

The *reflective Petri Nets* approach we have developed quite strictly adheres to the classical reflective paradigm [Cazzola, 1998]. It permits one to model an application and *separately* all its possible evolutions, and to dynamically adapt application model when evolution must occur.

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 P/T net of the software system prone to be evolved; whereas the second layer, called *meta-level* is represented by the *evolutionary meta-program*; in our case the meta-program is a WN composed by the *evolutionary strategies* that might drive the evolution of the base-level PN. We realistically assume that several strategies are possible at a given instant: one is selected in non-deterministic way. 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 in 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 that 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*, that 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 change to the influence area of the selected evolutionary strategy. The base-level PN afterward resumes. With respect to our previous work [Capra and Cazzola, 2005] this approach improves concurrency between levels.

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 Sect. 4 for seeing the benefits), which is updated only when necessary. So analysis/validation we can perform separately on both models, is not polluted by non relevant aspects.

3.1 Reflective Framework

A framework formalization in terms of WNs allows us to specify complex evolutionary patterns for the base-level PN in a simple, unambiguous way.

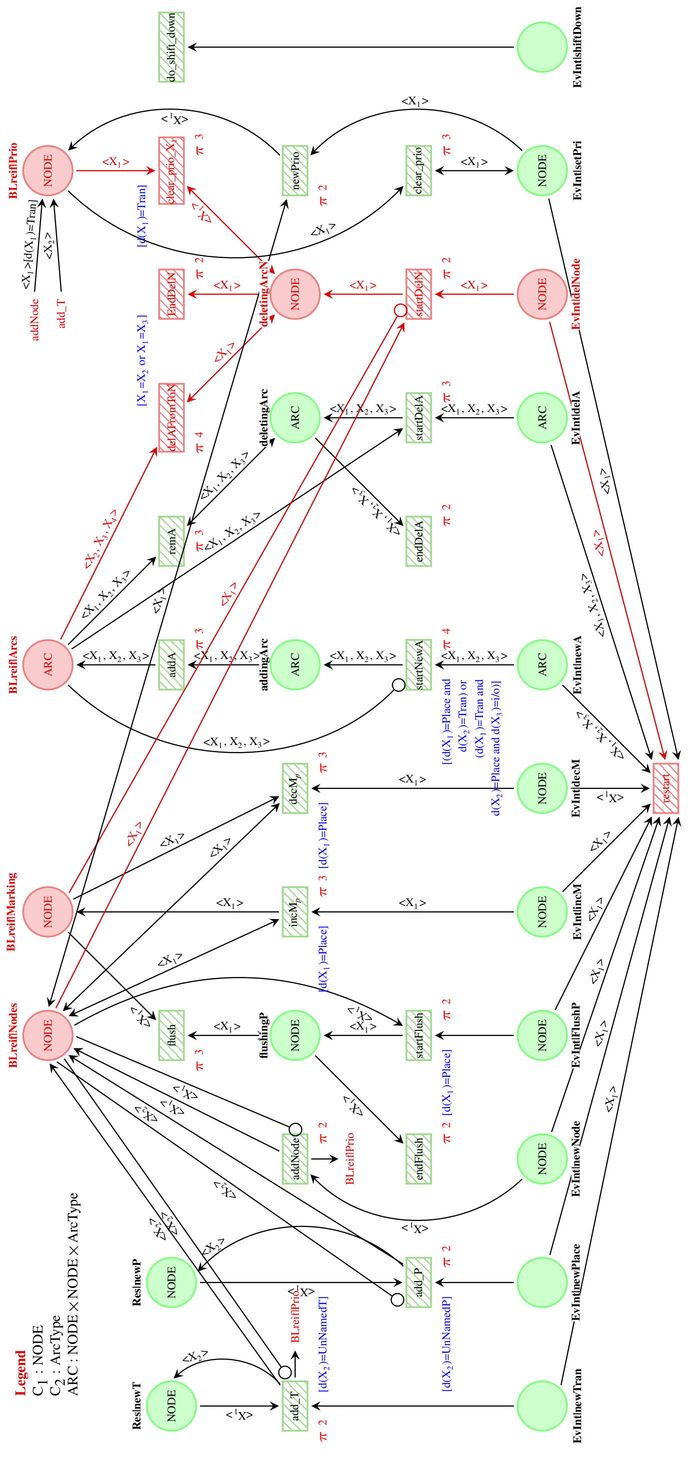


Figure 3: A detailed view of the framework implementing the evolutionary interface.

The reflective framework (figure 3) driven on the content of the evolutionary interface performs a sort of concurrent-rewriting on the base-level Petri net, suitably reified as a WN marking. Places whose labels have prefix `BLreif`[1] belong to the base-level reification (set $BLreif$), 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, that are bound to a given base-level PN.

3.1.1 Color Definitions

The framework basic color classes are $C_1 : \mathit{NODE}$, $C_2 : \mathit{ArcType}$. The definition of the structure of class NODE, in particular, takes place at system start-up. Letting $\mathcal{BL}$: $(P_b, T_b, W_b^+, W_b^-, H_b, \Pi_b, \mathbf{m}_0^b)$ be the base-level PN at system start-up, we have:

$$\begin{aligned}
\mathit{NODE} &= \underbrace{\mathit{Place}}_{\underbrace{P_b}_{p_1\cup\ldots p_n} \cup \underbrace{UnNamedP}_{x_1\cup\ldots x_k}} \cup \underbrace{\mathit{Trans}}_{\underbrace{T_b}_{t_1\cup\ldots t_m} \cup \underbrace{UnNamedT}_{y_1\cup\ldots y_r}} \cup\ \mathit{null} \\
\mathit{ArcType} &= i/o \cup h
\end{aligned}$$

The above three-level partitioning of class NODE (that collects the set of potential nodes of the base-level PN) into static subclasses may be considered as a default choice, that might be adapted/refined depending on modeling needs. Focusing on places, subclass P_b includes the base-level places, the only ones can be explicitly referred to by means of constants when programming an evolutionary strategy. For that purpose, it is further partitioned into singletons. Subclass *UnNamedP* instead contains places that might be added to the base-level without being explicitly named. It should be set large enough to be considered as a logically unbounded repository. Also the elements of this subclass can be referred to by means of constants, but only to make it possible an automatic updating of the base-level marking reification (as explained in the sequel). The structure of subclass *Trans* is analogously defined.

Class *ArcType* identifies two types of WN arcs, input/output and inhibitor. It is partitioned in two singletons.

The intuitive color domain definitions for the base-level PN reification are

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

given below:

$$\begin{aligned}\forall p \in BLreif \setminus \{\texttt{BLreif|Arcs}\}, \mathcal{C}(p) &: NODE\\ \mathcal{C}(\texttt{BLreif|Arcs}) &: ARC = NODE \times NODE \times ArcType\end{aligned}$$

3.2 Base-Level Reification

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 (section 3.5).

Definition 1. The reification of a base-level PN ($reif(\mathcal{BL})$) is the marking:

$$\begin{aligned}\mathbf{M}(\texttt{BLreif|Nodes}) &= \textstyle\sum_{n \in P_b \cup T_b} 1 \cdot n\\ \mathbf{M}(\texttt{BLreif|Prio}) &= \textstyle\sum_{t \in T_b} (\Pi_b(t) + 1) \cdot t\\ \mathbf{M}(\texttt{BLreif|Marking}) &= \textstyle\sum_{p \in P_b} \mathbf{m}_0^b(p) \cdot p\end{aligned}$$

$$\forall p \in P_b, t \in T_b \begin{cases}\mathbf{M}(\texttt{BLreif|Arcs})(\langle p, t, i/o\rangle) = W_b^-(p,t)\\ \mathbf{M}(\texttt{BLreif|Arcs})(\langle t, p, i/o\rangle) = W_b^+(p,t)\\ \mathbf{M}(\texttt{BLreif|Arcs})(\langle p, t, h\rangle) \quad = H_b(p,t)\\ \mathbf{M}(\texttt{BLreif|Arcs})(\langle t, p, h\rangle) \quad = 0\end{cases}$$

The evolutionary framework's colored initial marking ($\mathbf{M}_0$) is the reification of base-level PN at system start-up.

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 0, there will be the term $1\langle t_2\rangle$ in `BLreif|Prio`. The above three places represent the base-level topology: any change to their marking operated by the evolutionary strategy 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 inconsistency due to concurrent execution of several strategies is 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 Sect. 3.4.

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`) can logically hold only sets (in that case the reflective framework has in charge elimination of duplicates).

Being subject to changes the base-level reification needs to preserve a well-definiteness over the time. Let $\overline{m}$ be the support of multiset m, i.e., the set of elements occurring on m with multiplicity > 0.

Definition 2. $\mathbf{M}$ is well-defined if (n_1, n_2: $NODE$, k: $ArcType$)

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

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

Definition 3. Let $\mathbf{M}$ be well-defined. The associated base-level PN ($b\_level(\mathbf{M})$) is such that $P_b = Places \cap \overline{\mathbf{M}(\texttt{BLreif|Nodes})}$, $T_b = Trans \cap \overline{\mathbf{M}(\texttt{BLreif|Nodes})}$, $\forall p \in P_b, \mathbf{m}_0^b(p) = \mathbf{M}(\texttt{BLreif|Marking})(p)$, as concerns priorities $\forall t \in T_b$, $\Pi_b(t) = \mathbf{M}(\texttt{BLreif|Prio})(t) - 1$, finally arc functions W_b^-, W_b^+, H_b are set as in definition 1 (except for reading equations from right to left).

From definitions above it directly follows that $b\_level(reif(\mathcal{BL})) = \mathcal{BL}$. By the way $\mathbf{M}_0$ is well-defined. Through the algebraic structural calculus for WN introduced in [Capra et al., 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.

3.3 Evolutionary Framework Structure/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 the $BLreif$ interface, each one 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 command(s) issued by the evolutionary strategy sub-model. Every time a (multiset of) token(s) is pushed in one of these places, a sequence of immediate transitions is triggered that implements the corresponding command(s). A succeeding

Evolutionary Interface (the asterisk means that the marking must be 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 away 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.*	
Base-Level Reification (the asterisk means that the marking must be a set)	
`BLreif|Nodes`* *the content of this place represents the nodes used by the base-level Petri net.*	`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 used by the base-level Petri net.*	`BLreif|Prio` *the content of this place represents the priorities used by the base-level Petri net.*

Table 1: The evolutionary interface API and the base-level reification data structure.

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 corresponds to the type of command argument (either $NODE$ or ARC), except for `EvInt|newPlace`, `EvInt|newTran`, that 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". 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 $\texttt{clearPrio}_{X_1}$). Otherwise the command aborts and the whole meta-model composed by the reflective framework and the evolutionary strategy (see also section 3.5.3) 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 EvInt|delNode 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 (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.

3.3.1 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; that 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

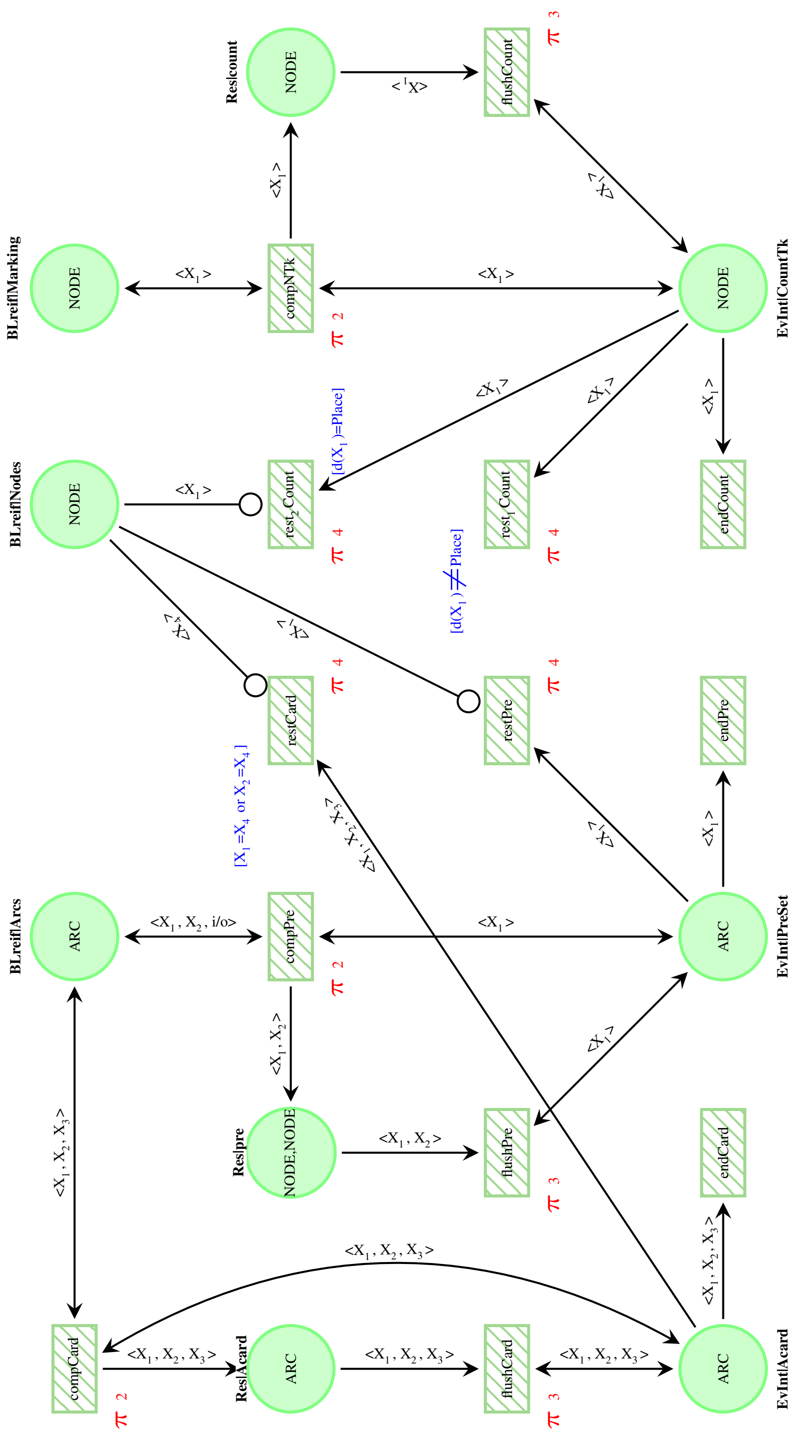


Figure 4: Basic introspection functions.

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 that the preset of node 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 `flush`).

3.4 The Evolutionary Strategy

The adopted model of evolutionary strategy (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), that 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. The language syntax is inspired by Hoare's CSP [Hoare, 1985] as concerns control structures (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 PN literature there are lot of examples of formal mappings from CSP-like formalisms (e.g. process algebras) to (HL)PN models (e.g. [Best, 1986] and more recently [Kavi et al., 1995]), from which we have taken inspiration.

General Schema

The evolutionary meta-program scheme corresponds to the CSP pseudo-code[2] in listing 1. 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

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

```
*[shift-up ? sh-up-occurred →
   [
     guard₁; event₁ ? event₁-occurred → strategy₁() □
     guard₂ → strategy₂() □
     true → strategy₃() □
                 ...
   ]
 ]
```

Listing 1: CSP code for the meta-program scheme

using "ad-hoc" language notations described in the sequel. Guard *true* means that the corresponding strategy might be always activated at every shift-up. A `guard` optionally ends with an input command that simulates occurrence of external events.

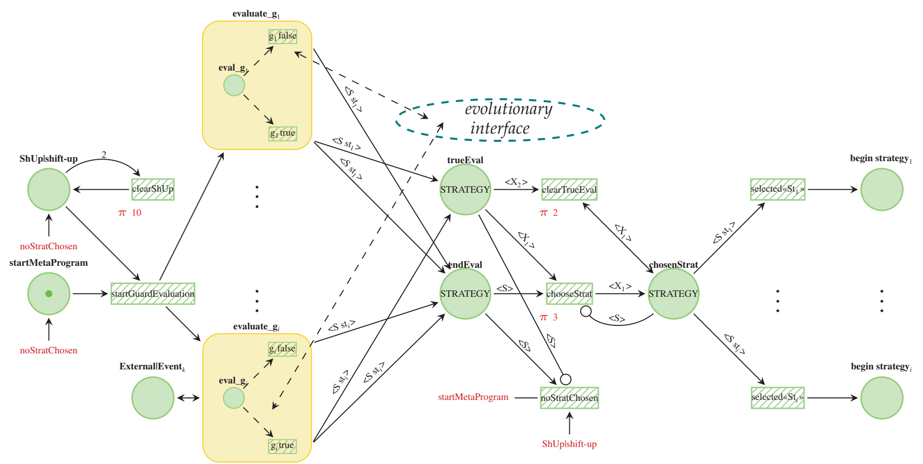
A more detailed view of this general schema in terms of PN is given in figure 5. Figure 5(a) shows the non-deterministic selection, whereas figure 5(b) shows the structure of i-th strategy. Color domain definitions are inherited from the evolutionary framework WN. An additional basic color class ($STRAT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

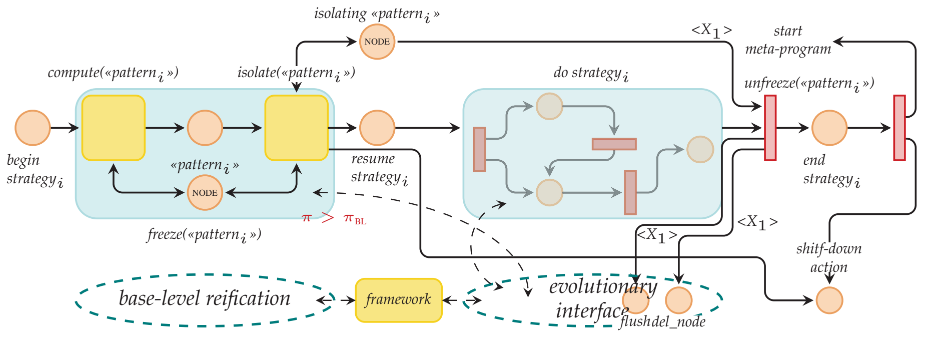
Occurrence of external events is modeled by putting tokens in particular "open" places (e.g. `External|event`$_k$ in figure 5(a)). The idea is that such places should be shared with other sub-models that simulate external event occurrence process. If one is simply interested in interactive simulation of the reflective architecture, he/she might think of such places as a sort of buttons to be pressed or released on demand.

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, that may be easily recognized in the picture.



(a) the strategy selection subnet



(b) the strategy structure

Figure 5: The meta-program generic schema.

It realizes a sort of *two-phases* approach: during the first phase (subnet-box `freeze(«pattern`$_i$`»)`) the meta-program sets the local influence area of the strategy, a portion of the base-level Petri Net reification that might be subject to changes. That area is expressed as a language "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 that 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, or that can change the marking of places belonging to it, are temporary inhibited from firing, until the strategy execution has terminated (the place `pattern*` holds the wider pattern image after this computation).

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 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 `do strategy`$_i$ in figure 5(b)), and the base-level resumes from the "suspended" state: that is implicitly accomplished by setting no dependence between the priority of `do strategy`$_i$ subnet transitions (arbitrarily assigned by the meta-programmer) and the priority of base-level PN transitions (in practice: setting the base-level PN's lowest priority equal to the priority level, assumed constant, of `do strategy`$_i$ subnet). The only constraint forced on `do strategy`$_i$ is that it can *exclusively* manipulate (by means of framework evolutionary interface) the nodes of base-level reification belonging to the pattern previously computed (this constraint is graphically expressed in figure 5 by an arc between the `do strategy`$_i$ 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. That reification does not involve the base-level area touched by the evolutionary strategy, that can continue operating without inconsistency. Before activating the final shift-down (that 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), that cannot be completely suspended while their evolution is being planned.

3.5 Casually connecting base-level and meta-program

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

3.5.1 Shift-up action

The shift-up action is realized for the first time at system start-up. The idea (illustrated in figure 6) is to connect in transparent, fully automatic way the base-

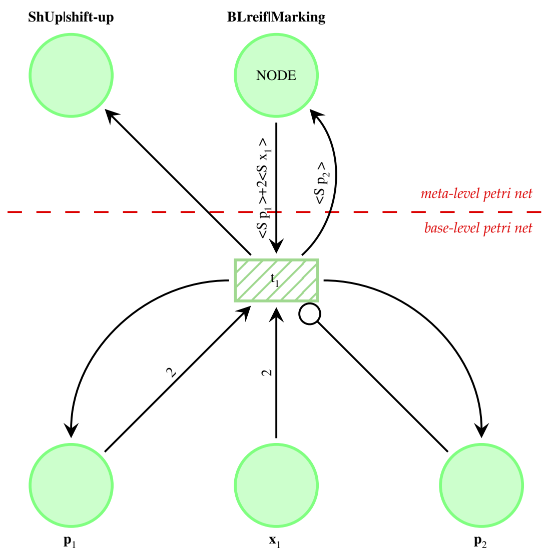


Figure 6: Reification implemented at PN level.

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 in the picture results in withdrawing one and two tokens from places p_1 and x_1, respectively, and in putting one in p_2. While token consuming 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 splitting of static subclass *UnNamedP* allows anonymous places introduced into the base-level (x_1) to be referred to by means of WN constant functions.

Transition t_1 signals its occurrence to the meta-program by putting one token in the uncolored interface-place `ShUp|shift-up`, that activates the meta-program itself (figure 5(a)).

3.5.2 Shift-down action

The shift-down action is the only operation that cannot be directly emulated at PN (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 PN 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 level within the whole reflective model, its occurrence replaces the current base-level PN with the PN described by the current reification, according to definition 3.

After a shift-down the base-level restarts from the (new) base-level initial marking, while the meta-program continues executing from its state preceding the shift-down.

3.5.3 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, PN-based approach. We are planning to extend the GreatSPN tool (designed for the WN formalism) to be used as simulation environment of our reflective architecture. To that purpose it must be integrated with a module implementing 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 et al., 2001] of GreatSPN. Following the model composition, absolute priority levels must be set, respecting the reciprocal constraints between components earlier discussed (e.g. framework lowest priority > meta-program highest priority). Finally, the whole model initial marking is set according to definition 1 as concerns base-level reification, putting colored token *null* in both places `Res|newP` and `Res|newT` (figure 3), and one token in place `startMetaProgram` (figure 5(a)).

3.6 Meta-Language Basic Elements

The meta-programming language disposes of four built-in types NAT, $BOOL$, $NODE$, $ArcType$ and of the **Set** and *Cartesian product* constructors. The arc ($ARC : NODE \times NODE \times ArcType$), the arc with multiplicity ($ArcM : ARC \times NAT$) and the marking ($Mark : NODE \times NAT$) are thus introduced as new types, in this way a multi-set can be represented as a set. Static subclass names (e.g., *Place*, *Tran*) 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 described in section 3.3. Their signatures are:

– $newNode(\mathbf{Set}(NODE)), newPlace(), newTran(), remNode(\mathbf{Set}(NODE))$;

– $flush(\mathbf{Set}(Place))$

– $addArc(\mathbf{Set}(ArcM))$, $remArc(\mathbf{Set}(Arc))$;

– $incMark(\mathbf{Set}(Mark))$, $decMark(\mathbf{Set}(Mark))$

– $setPrio(\mathbf{Set}(Tran))$

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

$$\texttt{*(e}_1\texttt{ in E}_1\texttt{, ..., e}_n\texttt{ in E}_n\texttt{)[command]}$$

makes «`command`» to be executed iteratively for each $e_1 \in E_1, \ldots, e_n \in E_n$; at each iteration, variables $e_1, \ldots, e_n$ are bound to particular elements of $E_1, \ldots, E_n$, respectively. If E_i is a color (sub-)class, then we implicitly refer to E_i 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 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 that allow the meta-programmer to specify patterns, i.e., (parametric) base-level portions that meet some requirements on structure/marking. The syntax for patterns and guards is shown in Grammar 1 in BNF form. The symbols: $pre(n)$, $post(n)$, $hset(n)$, $\#p$, $card(a)$, 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, respectively. They are translated into basic introspection commands (figure 4).

A pattern example is: $\{p : Place\,|\#p > \#p_1 \wedge isempty(pre(p) \cap hset(p))\}$, where p_1 is a constant, and p is a variable.

An example of guard is: $exists\ t : Tran|isempty(pre(t) \cup hset(t))$
(in the current version of the language not-nested quantifiers can be used).

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", that 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 (the box `isolate(«pattern`$_i$`»)` in figure 5(b)), which is one of the fixed parts of the meta-program, is given in Listing 2: 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,

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 | hinset
Pattern    ::= { variable InExpr | Guard }
Guard      ::= Predicate | LogOp variable InExpr Predicate | not( Guard ) |
               Guard BoolOp Guard
InExpr     ::= ε | in place | in NodeSet
LogOp      ::= exists | foreach
BoolOp     ::= and | or
```

† Terminals are in non-proportional font, non-terminals are in proportional font.

‡ Terms in represent elements whose meaning can be inferred from the model.

```
[
  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;
]
```

Listing 2: CSP code for the isolating pattern subnet

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 introduced inhibitor place at the end of the evolutionary strategy (see also figure 5(b)).

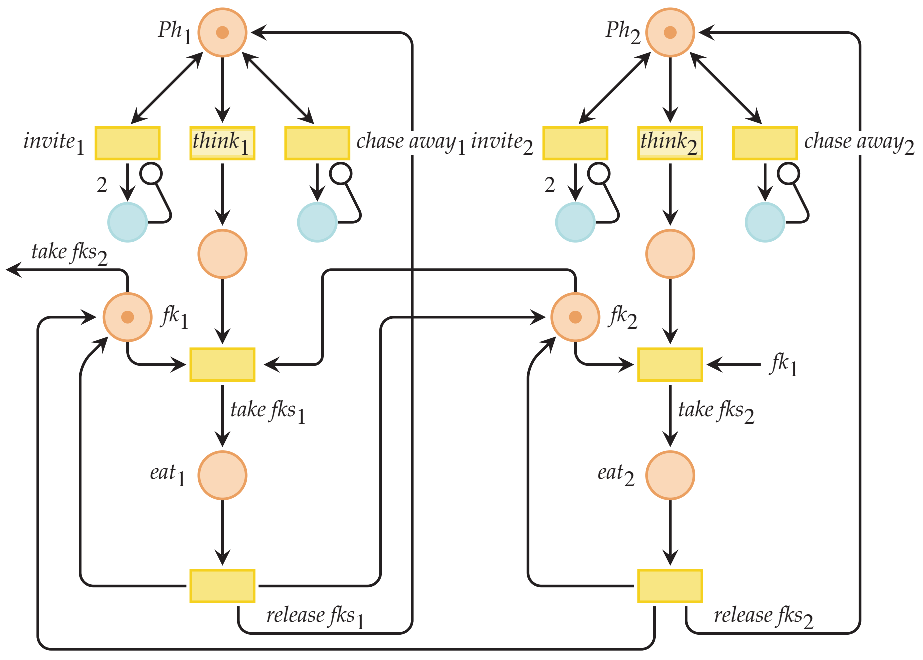


Figure 7: The base-level Petri net modeling the hurried philosophers problem with two sat philosophers and potentially infinite to sit.

4 Reflective Petri Nets in Action

To show our approach in action we consider as a case study a version of the *hurried philosophers* problem [Sibertin-Blanc, 2001], a variant of the well known dining philosophers problem that introduces an high dynamism and mobility degree: philosophers can join and leave on invitation the dining table. The version here considered has the following requirements:

- a number of philosophers (at least two) are initially seated at the same table;
- each philosopher can eat only when he/she contemporary gets the two nearest forks, according to the classical *dining philosophers* problem;
- philosophers at the table that are not eating have the following additional capabilities:
  - they can invite another philosopher (arbitrarily chosen) to join the table and sit down on either side if the table capacity has not been exceeded yet ;

  - they can ask one of the adjacent philosopher of leaving the table (if there are more than two philosophers sitting at the table);

– each philosopher is going around with his own fork; when he/she joins the table he/she keeps his/her fork, if he/she leaves the table he/she brings his/her fork.

This represents a simplification with respect to the *house of philosophers* presented in [Hoffmann et al., 2005], while preserving its efficacy. We have only a room with a table and philosophers that do not sit at the table are out of the system. The invitation and chasing away capabilities provoke the evolution of the net modeling the dining philosophers. It is necessary to reconfigure the whole net to add or remove a philosopher, this implies changing adjacencies and fork sharing.

Figure 7 shows the base-level PN for a minimal (or starting) system configuration, where two philosophers are sitting at the dinning table. The subnet representing the philosopher control flow is easily recognizable. The fk_i places represent forks, that are shared with the adjacent philosophers. Initially places Ph_i and fk_i are marked ($\mathbf{m}_0(Ph_i) = \mathbf{m}_0(fk_i) = 1$). The transitions labeled *invite* and *chase away* provide a very abstract representation, respectively, for invitation and chase away requests issued by a philosopher. An invitation is encoded by putting two tokens in a corresponding output place, while a “chase away” produces only one token. This makes it possible to distinguish these actions at meta-program level through introspection. After issuing an invitation or chase away, Ph_i continues running normally (thereby the self-loops from/to places Ph_i). The inhibitor arcs in figure 7 prevents accumulation of identical requests.

From figure 7, it should be fairly evident how simple is the design of our net, in particular if compared to the model proposed in [Hoffmann et al., 2005]. No details related to the net dynamic evolution are hard-wired in the base-level net. Evolution is delegated to the meta-program, that is automatically activated by the reflective framework when a philosopher is invited or chased away.

Let us describe the algorithm which is the core of the evolutionary strategy managing an invite action (Listing 3). We assume that one philosopher (`ph`) can invite someone else to sit down on either side. To that purpose, he/she disconnects itself from the adjacent fork (`fk`). The invited philosopher (`newph`) bears its own fork, that will be shared by `ph`, once he/she joins the table. If new philosopher's arrival were reflected on the base-level PN while `ph` is still eating, an inconsistent situation would arise due to the sharing of `newfk`: e.g., after `ph` finishes eating, place `newfk` might (inconsistently) hold two tokens. To avoid possible incongruence the evolutionary strategy is in charge of controlling (during the introspection phase) whether `ph` is eating or not, temporarily ignoring

```
[
 VAR ph, newph, fk, newfk, think, wait_fks, takes_fks, release_fks,
              ph1, wait_fks1, takes_fks1, isolating_pattern : NODE;
 VAR pattern, forks: SET(NODE);

 exists p in Place, #p > 1 → // strategy for invite action
    // begin pattern computation & isolation
    flush(p); // clearing place p
    ph = singleton(pre(pre(p))) ; // ph is the inviting phil.
    think = singleton(post(ph)\ pre(ph)); // transition "think"
    wait_fks = singleton(post(think)); // place "wait forks"
    take_fks = singleton(post(wait_fks)); // transition "take forks"
    eat = singleton(post(take_fks)); // place "eat"
    release_fks = singleton(post(eat )); // transition "release forks"
    forks = pre(take_fks)\ wait_fks; // forks of ph
     [ // ph is not eating: one of its forks is busy
        #eat = 0 and exists f in forks → fk = f; □
        #eat > 0 → break; // ph is eating
     ];
    pattern = {take_fks}; // take_fks is the only action to be frozen
    isolate(pattern); // isolating procedure and intermediate first shift-down
    // do_strategy begin: from now on the base-level resumes
    deleteArc({<fk,take_fks,i/o>, <release_fks,fk,i/o>}); // unhook fk,ph
    takes_fks1 = singleton(post(fk)); // identifying a phil. adjacent to ph
    wait_fks1 = singleton({px in pre(takes_fks1)\ fk|card(post(px))=1});
    ph1 = singleton(pre(pre(wait_fks1)));
      // a new philospher-net is created owning newfk
    newph = new-phil-net(newfk = newPlace(), fk);
      // ph is connected to newfk
    newArc({<newfk,take_fks,i/o,1>, <release_fks,newfk,i/o,1>});
    incMark({<newph,1>, <newfk,1>}); // places newph and newfk are marked
    flush(isolating_pattern); // influence area unfreezing
    delNode(isolating_pattern);
       // strategy for chase_away action
 □card({p | p in PHIL }) > 2 and
       (exists p in Place, #p == 1 and isempty(post(p)) →
    ...
]
```

Listing 3: CSP code for the invite strategy (body of repetitive command)

the invite request in the first case (a `break` instruction interrupts the current iteration of the embedding repetitive command). This is however not sufficient: transition `take_forks` (the only that can affect the influence area, constituted by place $\texttt{eating}_i$ in figure 7) must be temporarily inhibited (freezing procedure) to keep the model semantically consistent.

We cannot use constant symbols to refer to philosophers and forks since the evolutionary algorithm must be parametric and general (it starts up when any philosopher seating on the table issues an invite or a chase-away request). Once verified that the guard of the invite strategy is satisfied, a structural "pattern-matching" is operated to identify the philosopher (`ph`) that has issued the request. In a similar way its adjacent forks, and other nodes of interest (e.g. one adjacent philosopher), are identified.

The evolutionary strategy is carried out after the freezing/isolation of the transition `take_forks`. The strategy consists of detaching `ph` from one of adjacent forks `fk` (arbitrarily chosen between them), introducing a new philosopher-subnet through a corresponding function call (whose parameters are the new fork owned by the philosopher and `fk`, the function body is not indicated being fairly intuitive), finally connecting `ph` to the new philosopher's fork (`newfk`).

As concerns basic color classes of the corresponding WN model, with respect to the general schema described in section 3.1.1, a $PHIL$ subclass of $Place$ might be introduced to distinguish philosophers head-places (Ph_i).

4.1 Structural base-level analysis

The base-level PN may be analyzed using different techniques. Let us focus on structural techniques, that are elegant, sound, very efficient, but that in general are highly affected (and limited) by model complexity. Keeping evolutionary aspects separated from functional aspects encourages the use of structural techniques.

Operating e.g. the structural algorithms of GreatSPN tool, it is possible to discover place-invariants, as concerns our case-study (figure 7) they are of two well-known categories (subscript sum is modulo-n, n being the number of philosophers): $\forall \mathbf{m}, \forall i : 1 \ldots n$

$$\mathbf{m}(Ph_i) + \mathbf{m}(eat_i) + \mathbf{m}(wait\_fks_i) = 1,\ \mathbf{m}(eat_i) + \mathbf{m}(fk_{i+1}) + \mathbf{m}(eat_{i+1}) = 1$$

From that invariants a lot of properties descend: the model is live and bounded (places $\texttt{invite}_i$ and $\texttt{chase\_away}_i$ are structurally bounded), consequently the state-space contains a maximal home-space, and so on.

Since adjacencies between philosophers on the base-level PN are preserved by any evolutionary strategy (that might be simply argued looking at the meta-program code) these properties hold independently of system evolution.

4.2 A conventional model of hurried philosophers

To make a comparison with a traditional modeling approach, we have developed a model of hurried philosophers using classical Place/Transition nets, that is shown in figure 8. The GreatSPN format of the net in figure 8 can be downloaded from `http://homes.dico.unimi.it/~cazzola/dynamic-2of3.rar`.

Classical PNs have a fixed topology, so all dynamism/evolution of the system must be hard-wired in the model and bypassed when not in use. This task

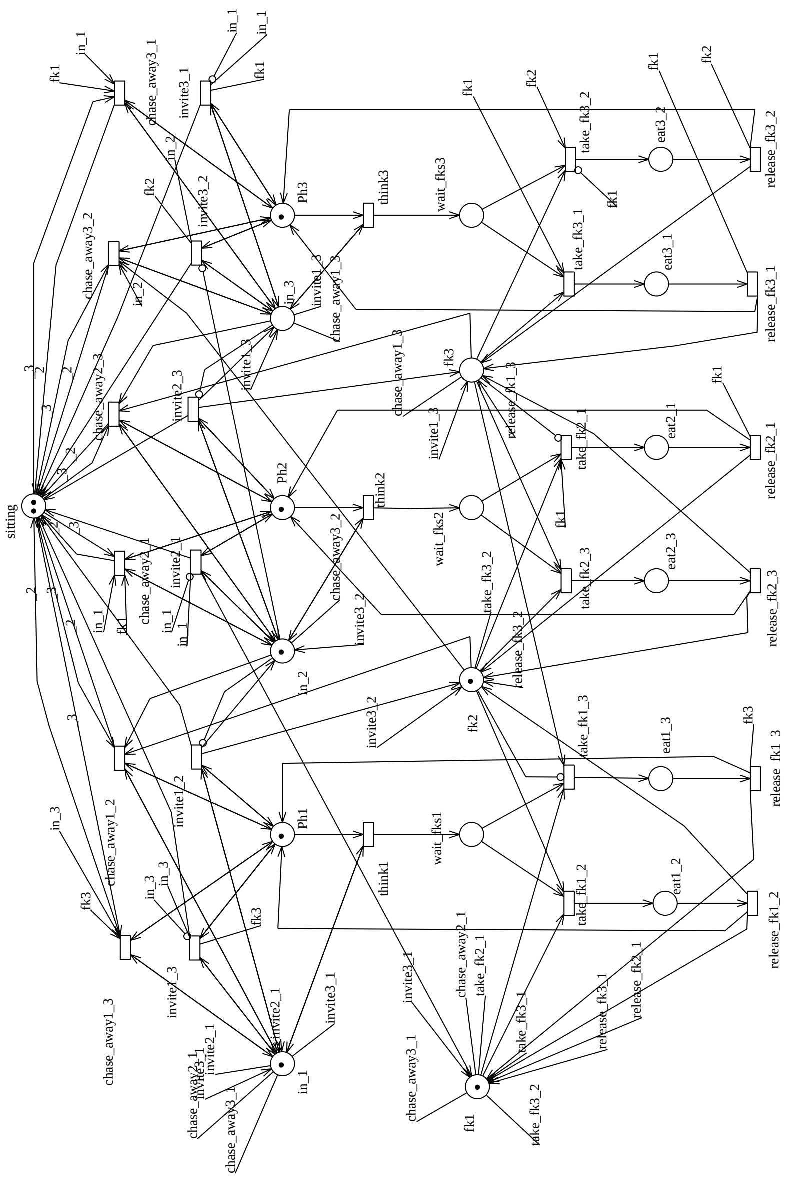


Figure 8: Classic Petri net modeling the hurried philosophers problem with two philosophers already dining and the possibility of adding one more philosopher at the most.

requires some expertise in PN modeling, and however might result in an incorrect (or partial) description of system behavior. What is worst, system analysis is polluted by a great deal of details that concern its evolution.

The intricacy of model in figure 8 is such that we skip any detailed description, focusing on a few essential aspects. First, the modeler has to fix the PN size according to the maximal (theoretical) system size (in our case, the table capacity). The net in figure 8 refers to a small table of size 3, where only two philosophers are currently seated. It corresponds to the situation modeled in figure 7.

The third philosopher appears in the picture despite he/she is currently out of the system. The state (*in/out*) of a philosopher is thus described by a place ($\mathtt{in}_i$). When the philosopher state is *in*, he/she can issue an "invite" to any *out* philosopher, or a "chase away" to a *in* adjacent philosopher. As usual a philosopher keeps its own fork either when he/she joins or leaves the table. The main difficulty consists of reproducing the correct forks adjacencies with respect to philosophers: when a philosopher eats, he/she must keep its own fork, and the "nearest" among the available ones. That implies splitting any action performed by a philosopher (but the "thinking" action) into a number of copies that depends on the number n of philosophers that could be in the system (for $n = 3$ actions are duplicated), and that is combinatorially growing.

The result is a model that is unreadable also for small system size. A model parametric in the table size could be defined using High-Level Petri Nets (e.g. CPN or WN), but in that case modeling skill becomes a requirement. What is relevant, however, and that does not depend on the adopted formalism, is represented by two critical aspects: liveness and boundedness (and other related properties) cannot be inferred by structural analysis, but only via state-space inspection, that however quickly reveals unfeasible as the system size grows. The other, probably more significant, aspect is that the model in figure 8 reproduces only in a small part the dynamism/mobility of the original system, where philosophers can freely move along the table.

5 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 (PN with inhibitor arcs are Turing-equivalent) however, classical PN are often considered unsuitable to model real systems. For that reason, several High Level PN (HLPN) paradigms (Colored PN, Predicate/Transition Nets, Algebraic Petri nets) [Jensen and Rozenberg, 1991] have been proposed in the literature 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 PN 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 PN 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 PN 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 paper) rely upon his paradigm.

Some quite recent proposals have extended Valk's original ideas. [Badouel and Darondeau, 1997] has introduced 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 and Oliver, 1998] defined 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 and 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 and 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 et al., 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 paper 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 the functional nets that 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 (HL)PN-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 and Oliver, 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 (HL)PN framework.

6 Conclusions and Future Work

Covering the evolutionary aspects of software systems has been widely recognized as one of the crucial challenges of modern software engineering. Many applications need to be updated or extended with new characteristics during their lifecycle. A good evolution has to pass through the evolution of the design information of the system itself.

PN are a central formalism for modeling concurrent and distributed systems. In this paper, we have faced the problem of the evolution of a PN model through the definition of a reflective architecture that allows the meta-program to observe and then to evolve the base-level PN. With this approach the model of the system and the model of the evolution (called evolutionary strategy) are kept separated, granting, therefore, the opportunity of analyzing the model without useless details. The evolutionary aspect is orthogonal to the functional aspect of the system.

The work here presented extends a previous one by the same authors, refining the evolutionary framework, and introducing a non-blocking mechanism for the base-level while an evolutionary strategy is in execution.

Ongoing research is in two different directions. We are planning to extend the GreatSPN tool for directly supporting our approach, both in the design and in the analysis phase. At the moment we are using two different formalisms for the base-level PN (ordinary PN) and the meta-level program (colored PN). In general it might be convenient to adopt the same formalism (may be algebraic PN) for both levels, this will give origin to the reflective tower allowing the designer to model also the evolution of the evolution of the system, and so on.

References

[Asperti and Busi, 1996] Asperti, A. and Busi, N. (1996). Mobile Petri Nets. Technical Report UBLCS-96-10, Università degli Studi di Bologna, Bologna, Italy.

[Badouel and Darondeau, 1997] Badouel, E. and Darondeau, P. (1997). Stratified Petri Nets. In Chlebus, B. S. and Czaja, L., editors, *Proceedings of the 11th International Symposium on Fundamentals of Computation Theory (FCT'97)*, LNCS 1279, pages 117–128, Kraków, Poland. Springer.

[Badouel and Oliver, 1998] Badouel, E. and Oliver, J. (1998). Reconfigurable Nets, a Class of High Level Petri Nets Supporting Dynamic Changes within Workflow Systems. IRISA Research Report PI-1163, IRISA.

[Bernardi et al., 2001] Bernardi, S., Donatelli, S., and Horvàth, A. (2001). Implementing Compositionality for Stochastic Petri Nets. *Journal of Software Tools for Technology Transfer*, 3(4):417–430.

[Best, 1986] Best, E. (1986). COSY: Its Relation to Nets and CSP. In Brauer, W., Reisig, W., and Rozenberg, G., editors, *Petri Nets: Central Models and Their Properties, Advances in Petri Nets (Part II)*, LNCS 255, pages 416–440. Springer, Bad Honnef, Germany.

[Cabac et al., 2005] Cabac, L., Duvignau, M., Moldt, D., and Rölke, H. (2005). Modeling Dynamic Architectures Using Nets-Within-Nets. In Ciardo, G. and Darondeau, P., editors, *Proceedings of the 26th International Conference on Applications and Theory of Petri Nets (ICATPN 2005)*, LNCS 3536, pages 148–167, Miami, FL, USA. Springer.

[Capra and Cazzola, 2005] Capra, L. and Cazzola, W. (2005). A Petri-Net Based Reflective Framework. In Arbab, F. and Sirjani, M., editors, *Proceedings of the IPM International Workshop on Foundations of Software Engineering (FSEN'05)*, Electronic Notes in Theoretical Computer Science 159, pages 41–59, Tehran, Iran. Elsevier.

[Capra et al., 2005] Capra, L., De Pierro, M., and Franceschinis, G. (2005). A High Level Language for Structural Relations in Well-Formed Nets. In Ciardo, G. and Darondeau, P., editors, *Proceeding of the 26th International Conference on Application and Theory of Petri Nets*, LNCS 3536, pages 168–187, Miami, USA.

[Cazzola, 1998] Cazzola, W. (1998). Evaluation of Object-Oriented Reflective Models. In *Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98)*, in 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium. Extended Abstract also published on ECOOP'98 Workshop Readers, S. Demeyer and J. Bosch editors, LNCS 1543, ISBN 3-540-65460-7 pages 386-387.

[Cazzola et al., 2004] Cazzola, W., Ghoneim, A., and Saake, G. (2004). Software Evolution through Dynamic Adaptation of Its OO Design. In Ehrich, H.-D., Meyer, J.-J., and Ryan, M. D., editors, *Objects, Agents and Features: Structuring Mechanisms for Contemporary Software*, Lecture Notes in Computer Science 2975, pages 69–84. Springer-Verlag, Heidelberg, Germany.

[Chiola et al., 1990] Chiola, G., Dutheillet, C., Franceschinis, G., and Haddad, S. (1990). On Well-Formed Coloured Nets and Their Symbolic Reachability Graph. In *Proceedings of the 11th International Conference on Application and Theory of Petri Nets,*, pages 387–410, Paris, France.

[Chiola et al., 1993] Chiola, G., Dutheillet, C., Franceschinis, G., and Haddad, S. (1993). Stochastic Well-Formed Coloured Nets for Symmetric Modelling Applications. *IEEE Transactions on Computers*, 42(11):1343–1360.

[Chiola et al., 1995] Chiola, G., Franceschinis, G., Gaeta, R., and Ribaudo, M. (1995). GreatSPN 1.7: GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Performance Evaluation*, 24(1-2):47–68.

[Farwer and Moldt, 2005] Farwer, B. and Moldt, D., editors (2005). *Object Petri Nets, Process, and Object Calculi*, Hamburg, Germany. Universität Hamburg, Fachbereich Informatik.

[Hoare, 1985] Hoare, C. A. R. (1985). *Communicating Sequential Processes.* Prantice Hall.

[Hoffmann et al., 2005] Hoffmann, K., Ehrig, H., and Mossakowski, T. (2005). High-Level Nets with Nets and Rules as Tokens. In Ciardo, G. and Darondeau, P., editors, *Proceedings of the 26th International Conference on Applications and Theory of Petri Nets (ICATPN 2005)*, LNCS 3536, pages 268–288, Miami, FL, USA. Springer.

[Hürsch and Videira Lopes, 1995] Hürsch, W. and Videira Lopes, C. (1995). Separation of Concerns. Technical Report NU-CCS-95-03, Northeastern University, Boston.

[Jensen and Rozenberg, 1991] Jensen, K. and Rozenberg, G., editors (1991). *High-Level Petri Nets: Theory and Applications*. Springer-Verlag.

[Kavi et al., 1995] Kavi, K. M., Sheldon, F. T., Shirazi, B., and Hurson, A. R. (1995). 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)*, pages 516–524, Kihei, Maui, Hawaii, USA. IEEE Computer Society.

[Kummer, 1998] Kummer, O. (1998). Simulating Synchronous Channels and Net Instances. In Desel, J., Kemper, P., Kindler, E., and Oberweis, A., editors, *Proceedings of the Workshop Algorithmen und Werkzeuge für Petrinetze*, volume 694 of *Forschungsberichte*, pages 73–78. Universität Dortmund, Fachbereich Informatik.

[Maes, 1987] Maes, P. (1987). Concepts and Experiments in Computational Reflection. In Meyrowitz, N. K., editor, *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, volume 22 of *Sigplan Notices*, pages 147–156, Orlando, Florida, USA. ACM.

[Odersky, 2000] Odersky, M. (2000). Functional Nets. In Smolka, G., editor, *Proceedings of the 9th European Symposium on Programming (ESOP 2000)*, LNCS 1782, pages 1–25, Berlin, Germany. Springer.

[Sibertin-Blanc, 2001] Sibertin-Blanc, C. (2001). The Hurried Philosophers. In Agha, G., De Cindio, F., and Rozenberg, G., editors, *Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets*, LNCS 2001, pages 536–538. Springer.

[Valk, 1978] Valk, R. (1978). Self-Modifying Nets, a Natural Extension of Petri Nets. In Ausiello, G. and Böhm, C., editors, *Proceedings of the Fifth Colloquium on Automata, Languages and Programming (ICALP'78)*, LNCS 62, pages 464–476, Udine, Italy. Springer.

[Valk, 1998] Valk, R. (1998). Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In Desel, J. and Silva, M., editors, *Proceedings of the 19th International Conference on Applications and Theory of Petri Nets (ICATPN 1998)*, LNCS 1420, pages 1–25, Lisbon, Portugal. Springer.

[Xu et al., 2003] Xu, D., Yin, J., Deng, Y., and Ding, J. (2003). A Formal Architectural Model for Logical Agent Mobility. *IEEE Transactions on Software Engineering*, 29(1):31–45.