A Petri-Net Based Reflective Framework for the Evolution of Dynamic Systems

Lorenzo Capra[1] and Walter Cazzola[2]

Department of Informatics and Communication
Università degli Studi di Milano
Milano, Italy

Abstract

Nowadays, software evolution is a very hot topic. Many applications need to be updated or extended with new characteristics during their lifecycle. Software evolution is characterized by its huge cost and slow speed of implementation. Often, software evolution implies a redesign of the whole system, the development of new features and their integration in the existing and/or running systems (this last step often implies a complete rebuilding of the system). A good evolution is carried out through the evolution of the system design information and then propagating the evolution to the implementation.

Petri Nets (PN), as a formalism for modeling and designing distributed/concurrent software systems, are not exempt from this issue. Several times a system modeled through Petri nets has to be updated and consequently also the model should be updated. Often, some kinds of evolution are foreseeable and could be hardcoded in the code or in the model, respectively.

Embedding evolutionary steps in the model or in the code however requires early and full knowledge of the evolution. The model itself should be augmented with details that do not regard the current system functionality, and that jeopardize or make very hard analysis and verification of system properties.

In this work, we 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 if necessary. Such an approach tries to keep the model as simple as possible, preserving (and exploiting) the ability of formally verifying system properties typical of PN, granting at the same time model adaptability.

Key words: Petri Nets, Reflection, Software Evolution.

[1] Email: `capra@dico.unimi.it`
[2] Email: `cazzola@dico.unimi.it`

©2005 *Published by Elsevier Science B. V.*

1 Introduction

Nowadays, software evolution is a very hot topic. Many applications need to be updated or extended with new characteristics during their lifecycle. Often, software evolution takes place without stopping the system and directly patching the software, i.e., without analyzing the situation and planning the evolution itself or foreseeing how the software could evolve at design-time before it really needs to evolve.

Software evolution, as well as software maintenance, is characterized by its huge cost and slow speed of implementation. Often, software evolution implies complete system redesign, development of new features and their integration in running systems (this last step often implies a complete system rebuilding).

A good evolution is carried out through evolution of system design information, and then through propagation of evolution to implementation. This approach should be the most natural and intuitive to use (because it adopts the same mechanisms adopted during the development phase) and it should produce the best results (because each evolutionary step is planned and documented before its application). Notwithstanding that its adoption is not so diffuse because it is not (or is only partially) supported by design/specification formalisms.

At the moment software evolution, independently of the adopted modeling techniques, is emulated by directly enriching original design information with properties and characteristics concerning possible evolutions. This approach has several drawbacks:

- planning evolution in advance means to have design information polluted by details related to the design of the evolved system (the model is confused because it does not represent a snapshot of the current system);
- evolution is not really modeled, its behavior is specified together with the behavior of the whole software and not as an extension that could be used in different contexts;
- planning evolution on demand means accessing to design information when necessary: often this is not possible (information about design are not available at run-time or they might not be released together with software) or difficult to realize (due to the growing system complexity, a long time might be required to understand how a system should correctly evolve).

PN, as a central formalism for modeling distributed (software) systems, are not exempted from these issues. PN are a powerful graphical formalism used to model discrete-event dynamic systems and to derive their properties. To correctly model systems able to evolve, without using tricks that require some PN expertise and that complicate dreadfully the model (affecting in many cases the possibility itself of analyzing it), it is necessary to enrich the PN modeling paradigm with evolutionary features.

At present, software evolution through evolutionary design is not sup-

ported by traditional PN classes. Normally it is achieved by merging the basic model of the system with information on the potential evolutions of the system itself. A similar approach pollutes the model with details not pertinent to the current structure of the software system. Pollution not only increases the complexity of the model but hinders the capacity of the existing tools of analyzing system properties without considering all the possible branches of evolution.

System evolution is an aspect orthogonal to the (current) system behavior, hence it could be subject to separation of concerns [7]. Separating evolution from the rest of the system is worthwhile, because evolution is made independent of the evolving system. If a given evolutionary plan should satisfy a general property (such as security, persistence, fault tolerance and so on), it could be (at least in part) reused from previous projects.

Separation of concerns could be applied also to a PN-based modeling approach. Normally system evolution takes place at a different time from design time. Design information (in our case, a PN modeling the system) will not be polluted by non pertinent details and will exclusively represent the system functionality without patches. This leads to a simplified and clean modeling approach by which a system can be analyzed without discriminating about what is its current structure and what might be the future one.

In this work 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. The proposed reflective framework gives both the mechanism for separating these aspects and the mechanism to merge them when a specific event occurs. With respect to several proposals recently appeared with similar goals, 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. This gives us the possibility of using existing tools and analysis techniques in a fully orthogonal fashion. The framework can be easily integrated to the GreatSPN tool [6], allowing sw 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 formalism and we survey the reflection parlance that we will be used in the rest of the paper; in section 3 we informally overview the framework, introducing the adopted terminology; in section 4 we present the components of the reflective PN framework; in section 5 we show our approach in action; finally in section 6 we draw our conclusions and perspectives.

2 Background

2.1 Petri Nets

We recall here only the basic notation and concepts about the formalisms used in the paper. We assume that the reader has some basic knowledge of high

level PN (HLPN) extensions [8].

The Well-formed Net (WN) formalism [5] was inspired by the Colored Petri Nets formalism and has the same modeling power of CPNs. Unlike CPNs, it includes transition priorities and inhibitor arcs. These features are very useful to represent the operational patterns of the meta-level models. As in all High Level PN formalisms, tokens in places are associated with an identifier (color), similarly transitions are parameterized, so that different (color) instances of a given transition can be considered for enabling and firing. However the WN color annotation syntax is peculiar: it was defined with the aim of developing efficient analysis techniques able to automatically exploit the behavioral symmetries embedded in the model.

Definition 2.1 A WN is a ten-tuple

$$\Big(\Sigma, P, T, \mathcal{C}, W^+, W^-, H, \Phi, \Pi, \mathbf{M}_0\Big)$$

P and T are the place and transition sets; transition input, output and inhibitor arcs are defined by W^+, W^-, H, which define also their color annotations (called arc functions); $\mathbf{M}_0$ is the initial (colored) marking. Π defines the transition priorities (assigning a priority level $\Pi(t) \in \mathbb{N}$ to each transition).

The other elements correspond to color annotations, described next. $\Sigma = \{C_1, \ldots, C_n\}$ is the set of basic color classes. Each C_i is a finite, non empty set of colors. Each color class can be partitioned into static subclasses $C_{i,1}, \ldots, C_{i,k}$, to distinguish some colors of the class. A basic color class may be circularly ordered, but we omit this part of the definition, being not used in our models.

$\mathcal{C}$ is a function associating a *color domain* to each place and transition. A color domain is defined as the Cartesian product of possibly repeated basic color classes, hence the color associated with tokens in place p as well as the *color instances* of a transition t, take the form of *tuples* of basic color class elements. The color domain of t is implicitly defined by the set of *variables* inscribing the arcs surrounding t ($Var(t)$), for simplicity assumed non empty. Each variable (in truth representing a *projection*) refers to a given basic color class. Letting $c(x_i)$ be the class of x_i, $\mathcal{C}(t)$ is defined as $c(x_1) \times \ldots \times c(x_m)$, $\forall x_i \in Var(t)$. A color instance of t ($\hat{c}_t$) is a consistent assignment of colors to variables in $Var(t)$.

An arc function $f(p, t)$ (where f may be W^+, W^-, H) is defined as $f : \mathcal{C}(t) \to Bag(\mathcal{C}(p))$; it takes the form of a weighted sum of tuples of (linear combinations of) *elementary functions* defined on basic color classes. The allowed elementary functions are the *projection*, selecting one element of a transition instance color tuple, and the *diffusion* function (denoted S or $S\,C_{i,j}$), returning the set of all elements in a given basic color (sub)class.

Φ is a T-indexed function, associating a guard to each transition, that restricts the set of admissible color instances to those satisfying a given predicate. Predicate basic terms represent simple conditions on transition instance tuple elements: equality of colors selected by projections (e.g., $x = (<>)y$),

or membership of a color to a specified static subclass (e.g., $d(x) = (<>)C_{i,k}$).

The transition priority Π is a T indexed function associating a priority level (in $\mathbb{N}$) to each transition; priority level 0 transitions are graphically represented as white boxes, while all other priority levels are for *immediate* transitions, graphically represented as black bars.

A marking $\mathbf{M}$ is a mapping $\mathbf{M} : P \to Bag(\mathcal{C}(p))$. A transition instance $\hat{c}_t$ *has concession* in $\mathbf{M}$ iff (i) for each input place p of t $W^-(t,p)(\hat{c}_t) \leq \mathbf{M}(p)$, (ii) for each inhibitor place p of t $H(t,p)(\hat{c}_t) >' \mathbf{M}(p)$, (iii) $\Phi(t)(\hat{c}_t) =$ true ($>, \leq$ relations and $+, -$ operations are implicitly extended to multisets; $>'$ restricts the comparison to non-zero elements of $H(t,p)(\hat{c}_t)$). A marking is called *tangible* if no immediate transitions are enabled, otherwise it is called *vanishing*.

$\hat{c}_t$ is *enabled* in $\mathbf{M}$ if it has concession, and no other higher priority transition instance has concession in $\mathbf{M}$. It can fire, leading to the new marking

$$\mathbf{M}'\colon \forall p \in P, \mathbf{M}'(p) = \mathbf{M}(p) + W^+(t,p)(\hat{c}_t) - W^-(t,p)(\hat{c}_t).$$

The class of nets we shall use for the *base-level* (see section 3) correspond to the uncolored version of WN: a *base-level* net is a is a $\mathsf{P/T}$ net with priorities and inhibitor arcs, formally a seven-tuple $\left(P, T, W^+, W^-, H, \Pi, \mathbf{M}_0\right)$, where $f(p,t) \in \mathbb{N}$ (f may be W^+, W^-, H), and $\mathbf{M}_0(p) \in \mathbb{N}$. The other definitions given above (concession, enabling, firing rule) are still valid, but for replacing $f(p,t)(\hat{c}_t)$ with $f(p,t)$.

2.2 Reflection

Computational reflection (or *reflection* for short) is defined as the activity performed by an agent when doing computations about itself [9]. This activity involves two aspects: *introspection* and *intercession*. Bobrow et al. define these two terms as follows:

> *Introspection is the ability of a program to observe and therefore reason about its own state. Intercession is the ability for a program to modify its own execution state, or alter its own interpretation or meaning* [2].

Reflection applies quite naturally to the object-oriented paradigm [9]. Just as objects in the conventional object-oriented paradigm are representations of *real world* entities, objects can themselves be represented by other objects, usually referred to as *meta-objects*. Computation done by meta-objects (*meta-computation*) is for the purpose of observing and modifying the objects they represent, called *referents*. Meta-computation is often performed by meta-objects by *trapping* the normal computation of their referents. In other words, an action of the referent is trapped by the meta-object, which performs a meta-computation either replacing or encapsulating the referent's action. Of course, meta-objects themselves can be manipulated by meta-meta-objects, and so on. Thus, a reflective system can be structured in multiple levels, constituting a *reflective tower*. Base-level objects (termed *base-objects*) perform

computations on the entities of the application domain. Objects in the other levels (termed *meta-levels*) perform computations on the objects residing in the lower levels. The interface between adjacent levels in the reflective tower is usually termed as *meta-object protocol* (MOP).

Reification is an essential capability of all reflective models. Each level of the reflective tower maintains a set of data structures representing (*reifying*) lower level computation. Of course, deciding which aspects are reified depends on the reflective model (e.g., structure, state and behavior, communication). In any case, the data structures comprising a reification must be *causally connected* to the aspect(s) of the system being reified. All changes to the reification are reflected in the system, and vice versa. Depending on the reflective model, the causal connection may operate at compile-, load- or run-time, but in all cases the meta-object programmer is not concerned about how the causal connection is achieved.

Transparency is another key feature of all reflective models. In the context of reflection, transparency refers to the fact that the objects in each level are completely *unaware* of the presence and workings of objects in higher levels. In other words, each meta-level is added to the base-level without modifying the referent level itself. An important application of transparency is in the separation of functional features from (possibly several distinct) nonfunctional features. Of course, *separation of concerns* enhances the system's modifiability. Depending on whether a required modification to the system involves functional or nonfunctional properties, functional objects alone or nonfunctional objects alone may be modified.

3 Reflective Petri Nets: the Model

The reflective approach we propose should permit the design of a software system formally described as a PN to dynamically evolve.

The approach is based on adopting a reflective architecture structured into two logical layers. The first layer, called *base-level*, is represented by the PN of the software system prone to be evolved, also called *base-level PN*; whereas the second layer, called *meta-level* (depicted in Fig. 1) is composed by the *evolutionary strategies* that will drive the evolution when certain events occur. The base-level PN and the PN representing the evolutionary strategies depend on the kind of evolution to carry out, and on the evolving system.

The reflective framework 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 (representing a generic PN), called *base-level reification*. The base-level reification is used by the evolutionary framework itself and by the evolutionary strategies to observe (introspection) and manipulate (intercession) the base-level PN. Each change on the reification will be reflected on the base-level PN by the framework, i.e., the base-level PN and

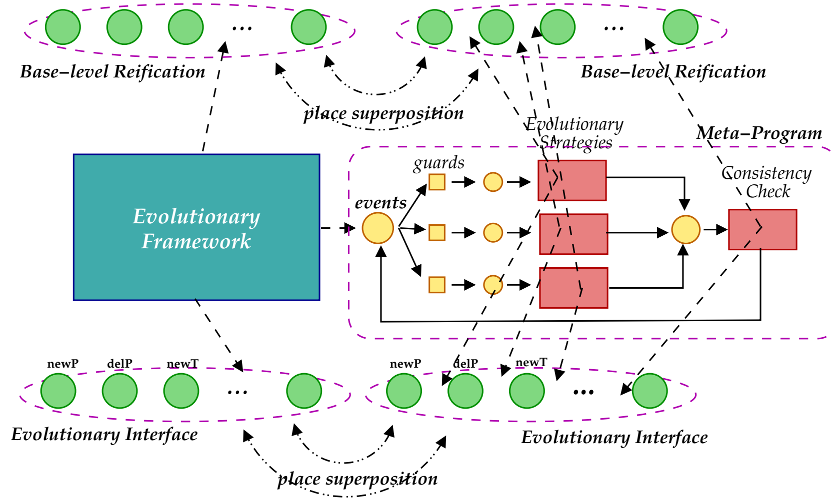


Figure 1. A snapshot of the meta-level. The base-level reification represents the connection between the two levels.

its reification are causally connected.

According to the reflective paradigm, the base-level PN evolves irrespective of the meta-level program (being not aware of the existence of a meta-level). The meta-level program is implicitly activated (shift-up action) under two conditions: either when the base-level PN model, during its simulation, reaches a given configuration, or following some external (i.e., not specified in the base-level) events. In such cases the base-level simulation is temporary suspended, and a suitable strategy is put into action.

Intercession on the base-level PN is carried out in terms of a set of basic operations suggested by the evolutionary strategy on the base-level reification. We have chosen a very minimal set of operations, called the *evolutionary interface*, that permits any kind of evolution both related to the structure and the behavior. The chosen set of operations allow us to introduce and to remove places, transitions and arcs, and to freely move the tokens all over the base-level net. This set of operations enables the meta-program to implement both structural and behavioral evolution (changing the topology and the marking of the base-level PN, respectively). A consistency check ends each strategy, with the aim of verifying preservation of basic system properties, before reflecting back changes on the base-level net (an example is given in Section 5.1). In a sophisticated design a recovery strategy might be executed in case of negative check result. Finally control passes back to the (evolved) base-level PN, that can continue its own dynamics.

A system must evolve to face an unpredictable event. In our framework, the occurrence of an event is modeled by a place. When an event occurs the evolutionary framework puts a token in this place. This action triggers the transitions that guard all the strategies. If a guard is verified the corresponding strategy is activated, that consequentially manipulates the base-level reification. Each evolutionary strategy is a PN that exploits the evolutionary

interface by putting a token in the place corresponding to the operation to trigger (see Fig. 1).

Evolutionary strategies have a *transactional* semantics: either they succeed, or leave the base-level PN unchanged. In our model, we realistically assume that several strategies are possible at a given instant: the adopted policy is to select one in non-deterministic way (see also Figure 1). A priority level can be also assigned to alternative strategies, to reduce (or annul) non determinism. These simple solutions, typical of the adopted PN-based approach, might be enhanced (without affecting the framework design) by adding a meta-meta-level having in charge selection of the "best" strategy.

The interaction mechanism between base-level and meta-level, and between meta-level entities, will be formalized in Section 4. Let us outline some essential aspects:

- the structure of the evolutionary framework is fixed, while the evolutionary strategy is related to a given base-level PN; both are modeled by WNs
- the evolutionary framework and the evolutionary strategy are separated components, sharing two disjoint sets of boundary places: the representation of the base-level PN and the evolutionary interface (Figure 1). Such interfaces represent the base-level (suitably reified as a *marking* which encodes topology and state of the base-level) and the evolutionary commands sent by a given strategy to the framework to be put into action, respectively. Interaction between components is realized through simple *place superposition*. This operator is supported by the `Algebra` module [1] of GreatSPN. Labels taking the form `place_name|postfix` denote boundary-places. Each pair of places with the same `postfix` are merged (by the way, they must have the same color domain);
- base-level reification place color domains are similar to formal parameters, that are instantiated when a given base-level PN is considered; their initial marking corresponds to the default base-level configuration
- implicit synchronization between base-level and meta-level might take place in several ways, specified at meta-level design phase. A reasonable assumption is that the meta-level is implicitly activated (that means a token is pushed in place *events* in Figure 1) whenever a new *tangible* marking of the base-level in entered (coherently with the assumption that base-level immediate transitions represent logical, or not observable, activities). In alternative, a set of (non immediate) transitions could be initially selected as responsible for meta-level activation (either when any of them is enabled, or after it has fired); this set might be dynamically changed by the meta-level itself. More sophisticated solutions are possible, but deeply discussing this topics is outside the scope of the paper
- once the meta-level is active, the base-level reification representing the base-level current marking is automatically updated. Base-level introspection is then carried out by the evolutionary strategy, checking for some condition

on base-level current state, topology, or both. For example it might be checked whether the base-level current marking is *dead*, or there is a set of places P' forming a *structural deadlock* (a potential source of deadlock: $\forall t \in T : \sum_{p \in P'}(W^{+}(p,t) - W^{-}(p,t)) < 0$).
Depending on introspection result, and/or occurrence of external events (represented by homonym boundary places) a particular strategy is selected and put into action. Occurrence of an external event could be simulated by (interactively) pushing a token in a given place, or it could be explicitly modeled by a separated PN suitably composed with the evolutionary strategy. Periodically occurring or scheduled events could be directly encoded in the base-level model (think e.g. to a city road whose access is forbidden to cars each week-end).

The framework is thus characterized by a fixed part (the high-level PN representing the evolutionary framework and the evolutionary interface), and by a part varying from time to time according to the evolving system and the necessary evolution (the base-level PN and the high-level PN representing the evolutionary strategy; the latter might in turn evolve, if a meta-meta-level were present). The fixed part is used to put evolution into practice for any kind of system, independently of its structure and behavior, is responsible for the reflective behavior of the architecture, and hides the work of the evolutionary sub-system to the base-level PN. This approach permits a clean separation between the PN describing the evolution and the model of the evolving system, that will be updated only when necessary. So the base-level PN model is not polluted by details related to evolution and the analysis we can perform on this model (as well as on the meta-level model) is not affected by pollution.

4 The Petri Net Based Reflective Framework

In this section the meta-level components of the PN-based reflective framework highlighted in Section 3 are described in detail. Their semantics, behavior and interactions are precisely stated.

Formalizing the evolutionary framework and the evolutionary strategy in terms of WNs allows us on one side to specify complex transformation patterns in a very simple way, on the other side to validate such models using consolidated analysis techniques. In particular a symbolic structural analysis technique (traditionally employed in classical PN) has been used, recently proposed for an extension of the WN formalism [3].

4.1 The Evolutionary Framework

The evolutionary framework model (Figure 2) defines the framework evolutionary engine. It performs a sort of concurrent-rewriting on the base-level, suitably reified as a marking of the evolutionary framework. Places whose labels have as prefix `BLreif` belong to the base-level reification, while those

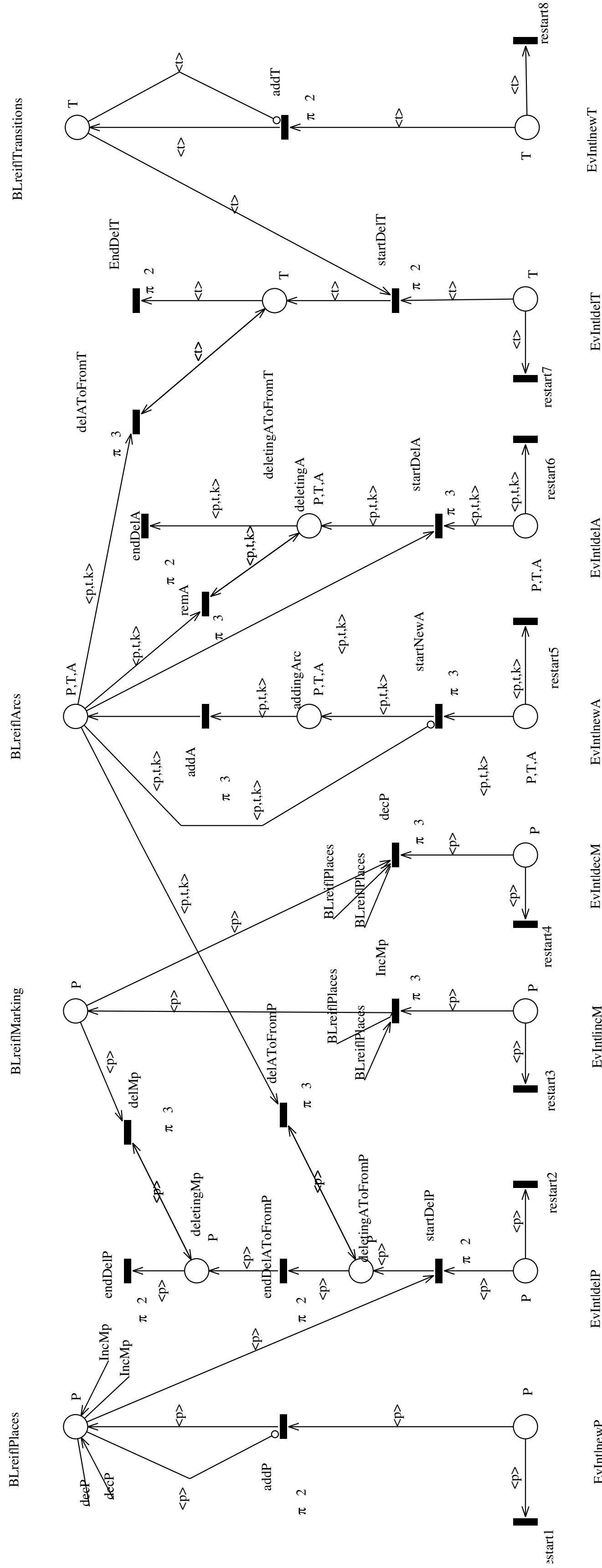


Figure 2: A detailed view of the framework implementing the evolutionary interface.

having as prefix `EvInt` belong to the evolutionary interface.

While WN structure and color annotations (color domains, transition guards, and arc functions) are generic, basic color classes and initial marking need to be instantiated (or, if you prefer to adopt a more reflective parlance, to be reified) for setting a link between evolutionary framework and base-level.

Letting $\left(P_B, T_B, W_B^+, W_B^-, H_B, \Pi_B, \mathbf{M}_0^B\right)$ be the base-level $\mathsf{P/T}$ net, the set of evolutionary framework WN basic color classes is $\Sigma_F = \{Place, Tran, ArcType\}$ (denoted P, T, A in Figure 2, respectively), where:

$$Place = P_B \cup otherP,\ Tran = T_B \cup otherT,\ ArcType = i \cup o \cup h$$

P_B and T_B are partitioned into cardinality one static subclasses identifying single places and transitions of the base-level standard configuration. Thus it is possible, when designing a given strategy, to explicitly refer to these elements. The subclasses $otherP$, $otherT$ contain (names of) places and transitions that might added to the base-level during its evolution. Even if WN classes are finite, $otherP$ and $otherT$ are to be logically considered as unbounded. This scheme of static partitioning of basic color classes might be further refined/adapted depending on modeling needs.

As concerns the colored initial marking of the evolutionary framework model ($\mathbf{M}_0^F$) we have (base-level reification color domains are intuitively set to: $\mathcal{C}(\texttt{BLreif|Places}) = \mathcal{C}(\texttt{BLreif|Marking}) : Place$, $\mathcal{C}(\texttt{BLreif|Trans}) : Tran$, $\mathcal{C}(\texttt{BLreif|Arcs}) : Place \times Tran \times ArcType$)

$$\mathbf{M}_0^F(\texttt{BLreif|Places}) = \sum_{p \in P_B} 1 \cdot p,\ \mathbf{M}_0^F(\texttt{BLreif|Trans}) = \sum_{t \in T_B} 1 \cdot t$$
$$\forall\ p \in P_B, t \in T_B, k \in ArcType :\ \mathbf{M}_0^F(\texttt{BLreif|Arcs})(\langle p, t, k\rangle) = f(p, t)$$

where $A(c)$ denotes the multiplicity of c in multiset A, and f denotes W_B^-, W_B^+, H_B depending on whether k is i, o, h, respectively.

The marking of places above corresponds to the topology of the base-level: for example, a base-level output arc of cardinality 2 from transition $t2$ to place $p1$, is encoded by a pairs of tokens $\langle p1, t2, o\rangle$ in `BLreif|Arcs`. Any change to the marking of these places corresponds to a modification that will be reflected on the base-level topology. The base-level current marking corresponds to the marking of place `BLreif|Marking`. At the beginning it is set to:

$$\forall p \in P_B,\ \mathbf{M}_0^F(\texttt{BLreif|Marking})(p) = \mathbf{M}_0^B(p)$$

Then it is automatically *refreshed* at any activation of the meta-level. By the way the evolutionary framework can modify this marking, meaning that the current state of the base-level is changed through reflection.

Evolutionary Framework behavior

The behavior associated with the evolutionary framework is very intuitive. Every place of the evolutionary interface represents a basic command that can be issued by the evolutionary strategy (described below). Each time a token is

pushed in that place, a sequence of immediate transitions is triggered putting into action the corresponding command. A succeeding sequence results in changing the base-level reification marking.

Eight basic actions are implemented: adding and removing a (set of) place(s) (`EvInt|newP`, `EvInt|delP`), adding and removing a (set of) transition(s) (`EvInt|newT`, `EvInt|delT`), adding and removing a (set of) arc(s) (`EvInt|newA`, `EvInt|delA`), and finally, increasing and decreasing a given marking (`EvInt|incM`, `EvInt|decM`). As an example, a token $\langle p \rangle$ occurring twice in place `EvInt|incM` corresponds to the command "increase the marking of p of two units". Commands may have side effects: for example removing an existing place p makes all the arcs connected to p are removed, and the tokens staying in p are flushed. For the sake of space, we have not included here commands for changing base-level transition priorities, and for adding a new place/transition without specifying its name.

A given command is carried out only if it is consistent with the current base-level configuration. Otherwise the command execution is aborted and the model is *restarted*, by means of special transitions ($\{\mathtt{restart}_i\}$) whose firing makes the whole meta-level model (obtained composing the evolutionary framework and evolutionary strategy) go back to its initial state (before last activation). The GreatSPN tool is provided with restart transitions. Trying to remove a yet not existing place/transition, trying to add an already existing place/transition, or trying to add a new arc $\langle p, t, k \rangle$ before p or t are present in the base-level topology are possible restart causes. Who designs the strategy is responsible for specifying a consistent sequence of commands. Some commands can be issued and put into action in parallel in a consistent way.

Any kind of net transformation can be defined as a combination of basic commands: for example "replacing the input arc connecting p and t with an inhibitor arc of cardinality three" correspond to putting one token $\langle p, t, i \rangle$ in `EvInt|delA`, and three tokens $\langle p, t, h \rangle$ in `EvInt|newA`. A more complex example will be shown in Section 4.

Different transition priority levels are used in order to guarantee *atomicity and consistency* of commands. In particular we have to remark that restart transitions are associated to the lowest priority, and that priority levels in Figure 2 are *relative*: they become absolute after composing the evolutionary framework model with the evolutionary strategy, so as the minimum priority used in the evolutionary framework is set greater that the maximum priority level used in the evolutionary strategy (according to the transactional semantics).

4.2 The Evolutionary Strategy

The evolutionary strategy specifies a set of alternative transformations on the base-level that can be put in action when some conditions (checked by introspection of the base-level PN) hold and/or some external event has occurred.

The general structure of the evolutionary strategy is depicted in Figure 1. The model color domains are the same as the evolutionary framework.

The evolutionary strategy specifies base-level transformation patterns of arbitrary complexity. A strategy designer is unaware of the details of WN formalism. So we define a minimal language that allows everyone to specify his own strategy in a simple way.

We have found convenient to adopt a syntax inspired to the Hoare's CSP, enriched with a few "ad-hoc" constructs and notations for easy manipulation of the base-level reification. A strategy specified in this way can be automatically translated into the corresponding WN model, that will be in turn composed with the evolutionary framework to obtain the desired meta-model (in the PN literature there are several examples of mapping from CSP-like programs into (HL)PN models).

Meta Language basic elements

The overall strategy scheme depicted in Figure 1 correspond to the following CSP program [3]

```
* [ cond_1; ext_event_1 ? event_1() --> strategy_1;cons_check_1 □
    cond_2; ext_event_2 ? event_2() --> strategy_2;cons_check_2 □
                  ...
    cond_n; ext_event_n ? event_n() --> strategy_n;cons_check_n
  ]
```

where $\texttt{cond}_i$ may be *true* (meaning that the corresponding strategy is always activated at every shift-up) and the input command that (for convenience) simulates occurrence of an external event is optional.

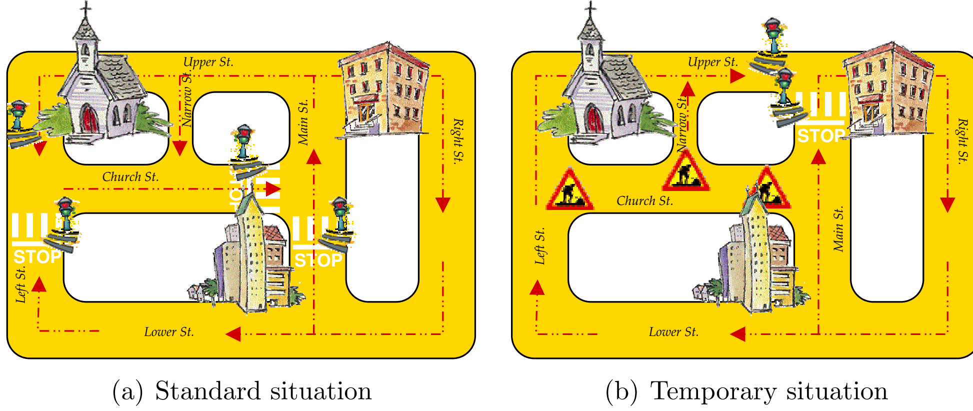
Other than four built-in types ($\mathbb{N}$, *bool*, *Place*, *Tran*) the language disposes of the *set* and *cartesian product* type constructors. The arc with multiplicity ($ArcM : Arc \times \mathbb{N}$) and the marking ($MarkP : Place \times \mathbb{N}$) types can thus be introduced (this way we can represent a multi-set as a set). New types can be also defined by using classical set operators.

The i-th strategy is defined in terms of routine calls corresponding to the set of basic actions described in Section 4.1. Their signatures are:

- $addPlace(\textbf{Set}(Place))$, $newPlace()$, $remPlace(\textbf{Set}(Place))$;
- $addTran(\textbf{Set}(Tran))$, $newTran()$, $remTran(\textbf{Set}(Place))$;
- $addArc(\textbf{Set}(ArcM))$, $remArc(\textbf{Set}(Arc))$;
- $incMark(\textbf{Set}(Mark))$, $decMark(\textbf{Set}(Mark))$

The meta-programmer can refer to (a set of) base-level elements either explicitly, by means of constant symbols (corresponding to static subclasses of *Place*

[3] 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.



(a) Standard situation (b) Temporary situation

Figure 3. City layout: a) the layout during normal activities b) the layout during road maintenance.

and $Tran$), or implicitly, by means of *variables* (normally used in membership test preceding loops or selection commands). By means of an assignment $p = newPlace()$ ($t = newTran()$) it is also possible to add an unspecified net element to the base-level, afterwards referred by variable p (t).

The following notations can be used to express any logical condition on the base-level topology/state ($e : Place \cup Tran$, $a : Arc$, $p : Place$): ${}^\bullet e$, $e^\bullet$, ${}^\circ e$, $conn(e)$, $\#p$, $card(a)$. Symbols ${}^\bullet e$, $e^\bullet$ denote the pre/post sets of a net element (defined as usual). Symbol ${}^\circ e$ denotes the set of elements connected to e via inhibitor arcs. Symbol $conn(e)$ denotes the set ${}^\bullet e \cup e^\bullet \cup {}^\circ e$. Last, #p denotes the current marking of p.

In addition to the CSP control structures, a particular version of the repetitive command can be used (letting E_i be any language type):

```
*(e1 in E1, ..., en in En)[ ≪command≫ ]
```

command is 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 $E_1, \ldots, E_n$.

Having at disposal the language above, it is possible to specify any kind of evolution, such as "for each place p belonging to the pre-set of t, if there is no inhibitor arc connecting p and t, add one with cardinality equal to the marking of p minus one", that becomes:

```
*(p in •t) [ card(<p,t,h>) == 0 --> addArc({<p,t,h,(#p - 1)>})]
```

5 Reflective Petri Nets in Action

5.1 Base-Level Model and Strategy Example

The simple example of base-level we are describing, taken from traffic management field, was presented in [4] (Figure 3). The corresponding PN model is depicted in Figure 4.

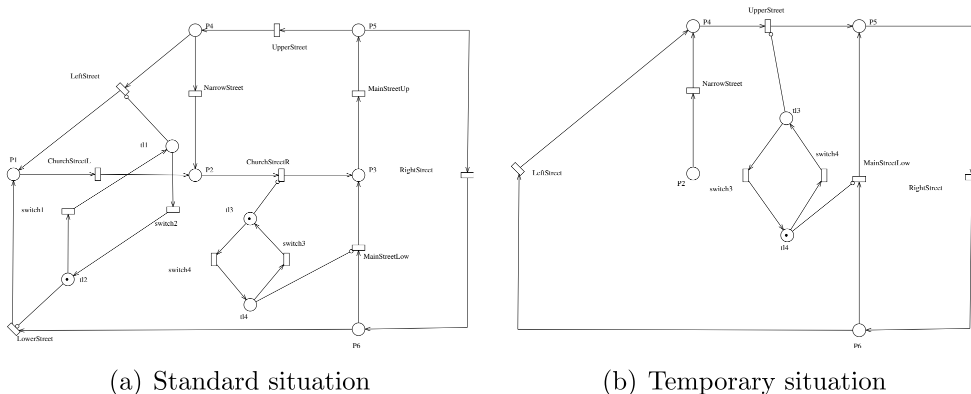
A small portion of a city map is considered. The entities of interest are cross-points, streets and traffic lights. The main scope of the evolutionary design is to guarantee safe connectivity between different points of the city, in front of occasional circumstances such as temporary or permanent closure of streets due to maintenance works, car accidents, natural events, and so on.

The adopted abstraction level does not consider other interesting aspects, such as the presence of vehicles having different priorities (e.g., emergency vehicles, and normal vehicles). Including also such aspects in the base model is only a descriptive matter (by the way the model complexity should be affected).

Mapping the city description into a PN model is straightforward. Net places represent cross-points, while transitions represent (mono-directional) flows of cars along street portions. If a given street can be run in both directions, then it will be associated with a pair of transitions. Finally, each traffic light is represented by a place (that when marked corresponds to the red signal) and a pair of transitions switching the traffic light color. Traffic lights are connected to the controlled cross-points through suitable inhibitor arcs. For safety reasons, some traffic lights must be synchronized (see the pairs of places $TL_1_2 : \{\texttt{tl1,tl2}\}$ and $TL_3_4 : \{\texttt{tl3,tl4}\}$ in Figure 4.a). In the base model the number of vehicles flowing through the city cross-points is initially unbounded: inhibitor arcs allow one to simulate traffic saturation in a very easy way.

A number of interesting properties can be derived by resorting to the powerful analysis techniques typical of PN (state-space exploration, structural analysis). For instance the model in Figure 4.a has nicely shown to be *live*, independently of the initial disposition of tokens representing vehicles along the city cross-points. That means each vehicle can always reach every destination, moving from any point of the city.

Assume that at a given moment the city map in Figure 3.a needs to be modified due to maintenance work at Church St., as described in 3.b. The corresponding evolutionary strategy is formalized by the WN models depicted



(a) Standard situation (b) Temporary situation

Figure 4. Petri Net model of the city map in Figure 3.

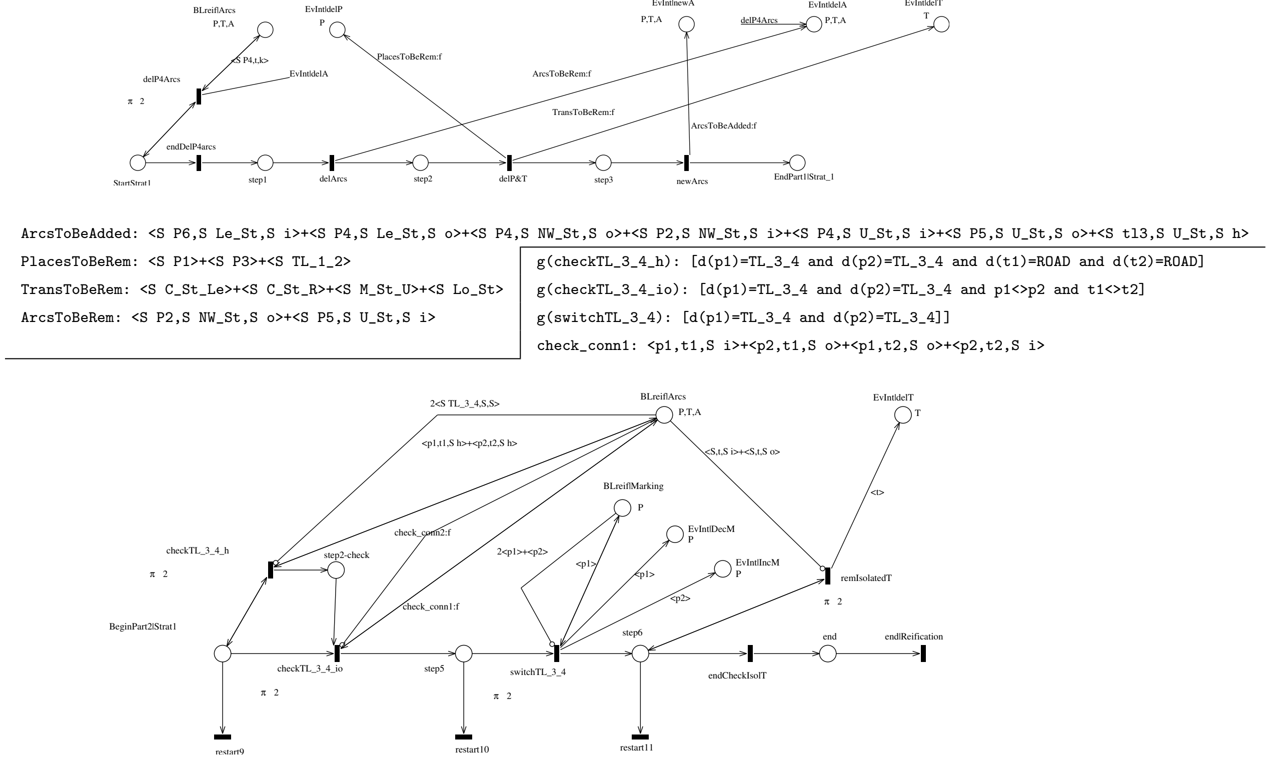


Figure 5. Evolutionary Strategy for the model in Figure 4.

in Figure 5. The upper part of the picture refers to the first phase of the strategy, and it corresponds to the meta-program fragment below (all names but t and k refer to constants; street names are shortened, for instance `C_St_Le` stands for `ChurchStreetLeft`):

```
ext_event ? work_at_church_st() -->
   *(t in conn(P4), k in ArcType)
           [card(<p,t,k>) <> 0 --> remArc({<p,t,k>})];
   remArc({<P2,NW_St,o>,<P5,U_St,i>});
   remPlace({P1,P3} + TL1_2);
   remTran({C_St_Le,C_St_R,M_St_u,Lo_St});
   addArc({<P6,Le_St,i>,<P4,Le_St,o>,<P4,NW_St,o,1>,<P2,NW_St,i,1>,
        <P4,U_St,i,1>,<P5,U_St,o,1>,<lt3,U_St,h,1>});
                        ...
```

The first command which is issued corresponds to: “safely remove all the arcs connected to P4”.

The second phase of the strategy is depicted in the lower part of 5 (the corresponding code fragment is not included for the sake of space). It realizes a sequence of high-level logical actions:

- the topological consistency of TL_3_4 synchronized traffic lights (which was touched in the first phase) is checked: traffic lights at the same carrefour must be reciprocally connected by a pair of input/output transitions (con-

necting arc must have multiplicity one); each traffic light must be linked to at least one street through an inhibitor arc of multiplicity one;

- the consistency of the state of TL_3_4 is checked (either traffic light $tl3$ or $tl4$ must be red at a given instant), and in case of positive check, it is switched (that corresponds to change the base-level marking);
- pending transitions are removed, to guarantee the semantics of the base-level (isolated transitions have always concession): the pairs of transitions $\{switch1, switch2\}$ connected to TL_1_2 turn out to be isolated, so they are automatically removed.

The strategy just described succeeds. After the evolution is reflected, the base-level model looks like in Figure 4.b.

6 Conclusions and Future Work

Nowadays, software evolution is a really hot topic. 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.

At the moment, to keep the presentation simple, we are using two different formalisms for the base-level PN (classic PN) and the meta-level program (colored PN). In general it might be convenient to adopt the same formalism (colored PN) for both the 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. In the next, we are going to extend the GreatSPN tool for supporting our approach.

References

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

[2] Bobrow, D. G., R. G. Gabriel and J. L. White, *CLOS in Context - The Shape of the Design Space.*, in: A. Pæpcke, editor, *Object Oriented Programming: The CLOS Perspective* (1993), pp. 29–61.

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

[4] Cazzola, W., A. Ghoneim and G. Saake, *System Evolution through Design Information Evolution: a Case Study*, in: W. Dosch and N. Debnath, editors, *Proceedings of the 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE 2004)* (2004), pp. 145–150.

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

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

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

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

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