Discrete Event Dynamic Systems (2021) 31:609–657
https://doi.org/10.1007/s10626-021-00343-3

Formal specification and verification of decentralized self-adaptive systems using symmetric nets

Matteo Camilli[1] · **Lorenzo Capra**[2]

Received: 30 March 2019 / Accepted: 25 May 2021
© The Author(s) 2021, corrected publication 2022

Abstract

Engineering distributed self-adaptive systems is challenging due to multiple interacting components, some of which monitor and possibly modify the behavior of managed components that operate in highly dynamic settings. Formalizing such systems having a decentralized adaptation control has been recognized as a hard task. In this article, we introduce a formal framework based on Symmetric Nets (a well-established subclass of Colored Petri nets) for modeling and analyzing distributed self-adaptive discrete-event systems. Even though Petri Nets represent a sound and expressive formal model of concurrency and distribution, they cannot specify in a natural way structural changes enacted by adaptation procedures. We overcome this limitation by means of a two-layer modeling approach that enables clear separation of concerns and allows multiple decentralized adaptation procedures to be specified, validated, and verified against formal requirements. Validation and verification techniques are supported by powerful off-the-shelf tools tailored to Symmetric Nets. A self-healing manufacturing system case study is used to show applicability, advantages, and shortcomings of the approach. In particular, complexity issues are thoroughly discussed and mitigated by adopting complementary approaches based on interleaving reduction and behavioral symmetries exploitation.

Keywords Self-adaptive systems · Self-healing systems · Distributed discrete-events systems · Petri nets

1 Introduction

Modern distributed discrete-event Systems (DESs) typically operate in dynamic environments and deal with frequently changing operational conditions. In particular, distributed

✉ Matteo Camilli
matteo.camilli@unibz.it

Lorenzo Capra
capra@di.unimi.it

[1] Faculty of Computer Science, Free University of Bozen-Bolzano, Bolzano, Italy

[2] Department of Computer Science, Università degli Studi di Milano, Milan, Italy

Published online: 05 August 2021

components may become temporarily or permanently unavailable, can disappear and appear, for instance due to faults and on-the-fly repairs. *Self-adaptation* – see de Lemos et al. (2013) – is an effective approach to deal with the increasing complexity and dynamism of these systems. *Self-adaptation* is often used as a means to achieve rapid adjustment of both production capacity and functionality, in response to new execution circumstances (operational contexts, environments and requirements), by reorganizing or changing its components (e.g., machines of the production systems, mechanisms for individual machines, sensors, controllers) directly in production. In this context there exists more than ever the need for robustness, resiliency, dependability. As stated by Weyns et al. (2013) and Arcaini et al. (2017), facing such a complexity in a distributed (decentralized) setting has been recognized as a major challenge in the field of self-adaptation. For this reason, formal methods providing the ability to reason on such a complexity are highly demanded. In particular, formal models representing both the structure and the behavior of self-adaptation and approaches to validate and verify them at design-time are of extreme importance to engineer self-adaptive systems.

Petri Nets (PNs) are a sound and expressive formal model of concurrency and distribution. However, both low-level and high-level PNs (even if Turing complete in some extensions) cannot represent in a natural way the ability of modifying their behavior and/or structure in response to their perception of the system itself as well as their surroundings and their goals. Several attempts to face this critical issue gave rise to new PN extensions, in which enhanced modeling power is not always accompanied by adequate analysis techniques. A representative example in this category is the “nets within nets” paradigm, introduced by Valk (2004). In this approach tokens represent themselves PNs. This ensures more flexibility in modeling, but significantly reduces the applicability of traditional PN analysis techniques.

To face this major challenge we propose a two-layer formal approach based on Symmetric Nets (SNs),[1] introduced by Chiola et al. (1993). Our approach gives to designers the ability to model distributed DESs in both their nominal and evolutionary behaviors, through a clear separation of concerns. The key idea, introduced in this article, is to exploit a “special” SN called *emulator* to encode, execute and change the behavior of any given P/T system. The design of the *managed* and *managing* subsystems are clearly separated. The modeler can specify the nominal behavior of the system and its own evolutionary behavior by means of *adaptation procedures* (one foreach adaptation concern). An adaptation procedure makes use of a collection of APIs, acting as *sensors* and *actuators* on the managed subsystem. Namely, they give the ability to read and modify the structure of the managed subsystem in order to achieve specific adaptation goals. Our modeling framework[2] allows behavioral symmetries to be exploited in order to analyze the symbolic state space in a efficient way. Furthermore, formal verification activities can take advantage of existing off-the-shelf software tools, such as GreatSPN introduced by Baarir et al. (2009).

A preliminary introduction of this ongoing research activity has been presented by (Capra and Camilli 2018). Here, we provide an extended presentation of the approach exemplifying both specification and verification using a self-healing manufacturing system case study. Self-healing (or self-repairing) refers to the ability of detecting faults/failures and recovering autonomously by changing the current configuration of the system. In this article, we

[1] SNs were previously known with the name Well-formed Nets. The new name was assigned after the formalism successfully underwent a standardization process.

[2] The framework and all the SN models presented in this article, including adaptation procedures and running examples, are publicly available at https://github.com/SELab-unimi/sn-based-emulator.

present: (*i*) a more accurate comparison with the state of the art; (*ii*) an improved definition and formal description of the framework; (*iii*) additional evaluation activities; (*iv*) and a new case study, i.e., a self-healing manufacturing system that shows adaptation behavior to achieve reconfigurability and fault tolerance. Specifically, the contribution of this article can be summarized as follows.

- we define a SN-based modeling framework for self-adaptive distributed DESs having decentralized adaptation control;
- we discuss the usage of symbolic structural analysis techniques to validate our modeling framework;
- we describe the applicability of our approach using a self-healing manufacturing system case study;
- we show how native SN analysis/verification techniques can be used to analyze systems modeled by using our framework.

The remainder of this article is as follows. In Section 2, we recall background notions of P/T nets and SNs. In Section 3, we present an example of self-healing manufacturing system used as main case study. In Section 4, we introduce an informal overview on our approach to model self-adaptive distributed systems using SNs. In Section 5, we formalize the emulating framework. In Section 6, we describe validation activities conducted on of the emulating framework. In Section 7, we describe how the modeler can specify adaptation procedures and how to compose them to create the whole system. In Section 8, we discuss complexity issues and how to tackle them. In Section 9, we describe formal verification of requirements. In Section 10, we discuss related work. Finally, in Section 11 we draw our conclusion and we outline future directions of our work.

2 Background

This section collects all the definitions used in the rest of the paper, and introduces the P/T net and SN formalisms. The reader may refer to (Reisig 1985) and (Chiola et al. 1993) for a detailed description of P/T nets and SN, respectively.

2.1 Multisets, Multiset-functions, and their operations

A *multiset* (or *bag*) over a domain D is a map $b : D \to \mathbb{N}$, where $b(d)$ is the *multiplicity* of d in b. The *support* $\overline{b}$ is $\{d \in D | b(d) > 0\}$: d is said an element of b ($d \in b$) if and only if $d \in \overline{b}$. A bag whose elements have multiplicity one is said *set-type*. A multiset b may be expressed as a weighted formal sum of its elements, where weights represent multiplicity (if equal to one it may be omitted). With some overloading, $\overline{b}$ may also denote the bag $\sum_{d \in b} d$. The *empty* multiset, i.e., the multiset with an empty support, is denoted $\emptyset_D$, or just $\emptyset$ if its domain is implicit.[3] The whole set of bags over D is denoted $Bag[D]$. Let $b_1, b_2 \in Bag[D]$. The *sum* $b_1 + b_2$, the *difference* $b_1 - b_2$, and the *intersection* $b_1 \cap b_2$ are bags in $Bag[D]$ defined, for any $d \in D$, as: $b_1 + b_2(d) = b_1(d) + b_2(d)$; $b_1 - b_2(d) = b_1(d) - b_2(d)$ if $b_1(d) \geq b_2(d)$, 0 otherwise; $b_1 \cap b_2(d) = min(b_1(d), b_2(d))$. Two bags b_1, b_2 are said *disjoint* if $b_1 \cap b_2 = \emptyset$. Associativity holds for $+$, $\cap$ (which may be treated as n-ary

[3] The same symbol denotes an empty set, letting the context to disambiguate.

operators), but not $-$. Relational bag-operators apply component-wise, e.g., $b_1 < b_2$ if and only if $b_1(d) < b_2(d), \forall d \in D$.

Let $k \in \mathbb{N}$, the *scalar* product $k \cdot b_1$ is $b'_1 \in Bag[D]$, s.t. $b'_1(d) = k \cdot b_1(d), \forall d \in D$.

Let $b_i \in Bag[D_i]$, $i : 1 \ldots \mathrm{n}$. The *Cartesian* product $b_1 \times b_2 \times \ldots b_\mathrm{n}$ is the bag $b' \in Bag[D_1 \times D_2 \times \ldots D_\mathrm{n}]$ defined as:

$$b'(\langle d_1, d_2, \ldots, d_\mathrm{n} \rangle) = b_1(d_1) \cdot b_2(d_2) \cdot \ldots b_\mathrm{n}(d_\mathrm{n}), \forall d_1 \in D_1, \ldots, d_\mathrm{n} \in D_\mathrm{n} \tag{1}$$

Bag-operators have intuitive functional extensions. Let $f_1, f_2 : D \to Bag[D']$, and *op* be a binary operator[4]: $f_1 \; op \; f_2 \; : D \to Bag[D']$ is $f_1 \; op \; f_2 \; (d) = f_1(d) \; op \; f_2(d)$, $\forall d \in D$. Analogously, $\overline{f_1} : D \to 2^{D'}$ is $\overline{f_1}(d) = \overline{f_1(d)}$. The support is a bridge between bag- and set-functions. As for relational operators, $f_1 < f_2$ if and only if $f_1(d) < f_2(d)$, $\forall d \in D$. The symbol $\emptyset_{D,D'}$, or $\emptyset$ if the arity is implicit, denotes the function $D \to \emptyset_{D'}$, $\forall d \in D$. Function equivalence is naturally set as $f_1 \equiv f_2$ if and only if $f_1(d) = f_2(d)$, $\forall d \in D$. The notions of *set-type* and *disjoint* extend to bag-functions as well, considering all the elements of function(s) domain(s).

Let $f_i : D \to Bag[D_i]$. The *scalar product* $k \cdot f_i$ is $k \cdot f_i(d), \forall d \in D$. The function *Cartesian product* $f_1 \times f_2 \times \ldots f_\mathrm{n} : D \to Bag[D_1 \times D_2 \times \ldots D_\mathrm{n}]$ is defined as:

$$f_1 \times f_2 \times \ldots f_\mathrm{n}(d) = f_1(d) \times f_2(d) \times \ldots f_\mathrm{n}(d), \forall d \in D \tag{2}$$

The notation $\langle f_1, f_2, \ldots, f_\mathrm{n} \rangle$, called*function-tuple*, is used in place of $f_1 \times f_2 \times \ldots f_\mathrm{n}$. Two operators are peculiar to bag-functions. Let $f : D \to Bag[D']$. The *transpose* $f^{Tr} : D' \to Bag[D]$ is $f^{Tr}(x)(y) = f(y)(x), \forall x \in D', y \in D$. The *linear extension* $f^* : Bag[D] \to Bag[D']$ is $f^*(b) = \sum_{x \in b} b(x) \cdot f(x), \forall b \in Bag[D]$. Function composition builds on linear extension. Let $h : D^{primeprime} \to Bag[D]$, then $f \circ h : D^{primeprime} \to Bag[D']$ is $f \circ h(d) = f^*(h(d)), \forall d \in D^{primeprime}$. We use the same notation (f) for a function and its linear extension, implicitly referring to the latter when the argument is a bag.

Finally, let $\{f_i\}$ be a family of functions $D \to Bag[D']$. A *linear combination* $F = \sum_i \lambda_i \cdot f_i, \lambda_i \in \mathbb{Z}$, is a function $D \to Bag[D']$ if and only if $\forall d \in D, x \in D'$: $F(d)(x) = (\sum_i \lambda_i \cdot f_i(d)(x)) \geq 0$. In that case, F is called *well-defined.*

2.2 Place/Transition (P/T) nets with inhibitor arcs

A P/T net enriched with inhibitor arcs is a 5-tuple (P, T, I, O, H), where:

- P, T are non-empty, finite sets such that $P \cap T = \emptyset$
- I, O, H are functions $P \times T \to \mathbb{N}$, i.e., $\{I, O, H\} \subset Bag[P \times T]$
- every element of $P \cup T$ occurs in I, O, or H.

The elements of P and T are called *places* and *transitions*, respectively. The former, drawn as circles, represent system state-variables, whereas the latter, drawn as bars, represent state local changes. The overall (distributed) state of a P/T net, called *marking*, is defined as a bag $m \in Bag[P]$. The marking of p is $m(p)$.

A P/T net is a kind of directed, bipartite multi-graph, with *input*, *output*, *inhibitor* edges (the latter, drawn with an ending small circle), described by I, O, H, respectively. Let $f \in \{I, O, H\}$: if $f(p,t) = k,\; k > 0$, then a corresponding weight-k arc connects p to t. The assumption that there is no *isolated node*, i.e., a node that no edge is incident

[4] Operators are implicitly overloaded.

to, is more than acceptable from the modeling point view and plays an important role in a reconfigurable context.[5]

The behavior of a P/T net in a given marking is specified by the transition *firing rule*. A transition $t \in T$ is *enabled* in a marking m if and only if:

$$\forall p \in P : \; I(p,t) \leq m(p) \wedge (H(p,t) \neq 0 \Rightarrow H(p,t) > m(p)). \tag{3}$$

If t is enabled in m then it may fire, leading to marking m', where:

$$\forall p \in P : \; m'(p) = m(p) + O(p,t) - I(p,t). \tag{4}$$

This is denoted $m[t\rangle m'$.

A pair (N, m), where N is a P/T net and m is a marking of N, is called P/T system. The interleaving semantics of a P/T system (N, m_0), where m_0 represents the initial state, is specified by the *reachability graph* (RG), an edge-labelled, directed multi-graph (V, E) whose nodes are markings. The RG is defined inductively: $m_0 \in V$; if $m \in V$ and $m[t\rangle m'$, $m' \neq m$, then $m' \in V$ and $m \xrightarrow{t} m' \in E$. The set V is called reachability set.

2.3 Symmetric nets

Symmetric Nets (SNs),[6] introduced by Chiola et al. (1993), are a high-level Petri net standard[7] formalism featuring a particular syntax which highlights the behavioral symmetries of systems. SNs are a flavor of *Colored* Petri nets, introduced by Jensen (1997), that have two different types of transitions, for *observable* (time-consuming, in stochastic SN) and *immediate* (logical, non-observable) events. The latter are drawn as tiny bars and take priority over the former type. In this section, we formally define SNs using the net in Fig. 1 to illustrate the base elements of the syntax. Figure 1 represents an excerpt of a simple broadcast communication protocol among nodes of a network. A node of the network (logically partitioned in two sub-networks) broadcasts messages to the other nodes. Upon reception of a data-message, a node broadcasts in turn an ack to the others. An ack is kept by the initial sender and discarded by the remaining nodes. A session successfully ends if the initial sender receives ack replies by all the others. Messages sent between the sub-networks may get lost but the excerpt here shown does not handle this event.

2.3.1 SNs and Color-annotations

A SN is a 9-tuple $(P, T, \mathcal{C}, \mathcal{D}, g, \mathrm{I}, \mathrm{O}, \mathrm{H}, \pi)$, where P and T are the finite, disjoint sets of places and transitions. $\mathcal{C} = \{C_i, i = 1 \ldots, \mathrm{n}\}$, $\mathrm{n} \in \mathbb{N}^+$, holds the *color classes*, which define the SN color structure: they are finite, pair-wise disjoint sets, each representing (at least two) system entities of a certain kind. A color class C_i *may* be either *circularly ordered* or *partitioned* into *static subclasses* $C_{i,j}$. The static partitioning of classes determines the symmetry of a SN model, indeed, all and only the colors of a subclass denote homogeneous components with an equivalent behavior. For instance, the SN in Fig. 1 has two color classes N, L, representing network nodes and (types of) messages, respectively, each partitioned in two static subclasses. The elements of each subclass of N (representing a sub-network) are symmetric, i.e., undistinguishable from one another. The two subclasses of L are singletons. Each SN model is also provided with a *neutral* color class $\mathrm{X} = \{\bullet\}$.

[5] In an ordinary context, it might be further constrained to I, O edges

[6] Originally equipped with a stochastic semantics and known as *Well-formed Nets*.

[7] ISO/IEC documentation available at https://www.iso.org/standard/43538.html.

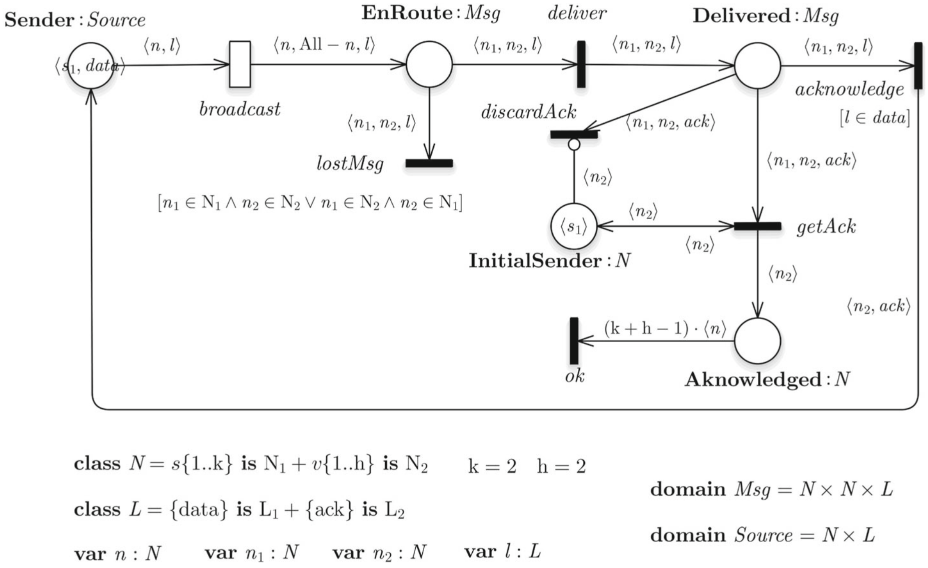


Fig. 1 SN example of a simple broadcast protocol (annotations follow the syntax of the GREATSPN graphical interface)

$\mathcal{D}$ maps each $v \in P \cup T$ to a *color domain*, usually defined as a Cartesian product of color classes: $\mathcal{D}(v) = \prod_{C_i \in \mathcal{C}'} C_i^{e_i}$, where $\emptyset \subset \mathcal{C}' \subseteq \mathcal{C}$, and $e_i \in \mathbb{N}^+$ is the number of repetitions of C_i in the domain. There may also be nodes with a neutral domain $\mathcal{D}(v) = \mathrm{X}$. A place color domain $\mathcal{D}(p)$ defines the possible colors (tuples of color elements from $\mathcal{D}(p)$) of the tokens which can stay in p. For instance, the color domain of the place `EnRoute` is $\mathrm{N}^2 \times \mathrm{L}$. A transition color domain $\mathcal{D}(t)$ defines the possible *firing instances* of t: these are tuples of color elements, and typed local variables ($Var(t)$) are used to refer to such elements in a tuple in $\mathcal{D}(t)$: for each $C_i \in \mathcal{C}'$, there are e_i distinct type-C_i variables. Thus $Var(t)$ implicitly defines $\mathcal{D}(t)$. A transition instance may be intuitively seen as a binding of colors to transition's variables. An instance of transition `broadcast` (the model's only observable transition), with $\mathcal{D}($`broadcast`$) = \mathrm{N} \times \mathrm{L}$, e.g., is a color binding of variables n and l. At the beginning, the place `Sender` holds a single token $\langle s_1, data\rangle$, in that case the only possible binding for `broadcast` is $(n = s_1, l = data)$. Formally, transition variables are interpreted as *projection* functions (Section 2.3.3).

A transition t with $Var(t) = \emptyset$, or a place p that holds undistinguishable tokens, has a neutral domain. For simplicity, the arcs that a neutral SN node is incident to are inscribed by $\mathbb{N}^+$ values. Similarly, the marking of a neutral place p is expressed by a value in $\mathbb{N}$.

g maps each $t \in T$ to a *guard* $g(t) : \mathcal{D}(t) \to \{true, false\}$, formally defined in Section 2.3.3. A guard *restricts* a transition's instances. A transition instance $b \in \mathcal{D}(t)$, also denoted (t, b), is said *valid* if and only if $g(t)(b) = true$. The color domain of t actually refers to the *restriction* of $\mathcal{D}(t)$ to valid instances of t (assumed non-empty). For simplicity, we use the same symbol. When omitted, a transition guard is meant equal to the constant $true$.

$\mathrm{I}[p, t]$, $\mathrm{O}[p, t]$, $\mathrm{H}[p, t]$, are families of functions, $\mathcal{D}(t) \to Bag[\mathcal{D}(p)]$, defined for each pair $(p, t) \in P \times T$, annotating input/output/inhibitor arcs respectively. The syntax of an arc function is formally defined in Section 2.3.3.

Finally, π is a map $T \rightarrow \mathbb{N}$, and $\pi(t)$ is the priority of t. A transition t is said *observable* if and only if $\pi(t) = 0$, otherwise t is said *immediate*. All the transitions in Fig. 1 but `broadcast` are immediate and have the same priority.

2.3.2 Semantics

A SN *marking* $\mathbf{m}$ is a P-vector such $\mathbf{m}[p] \in Bag[\mathcal{D}(p)]$. We call $\mathbf{m}[p]$ the marking of p, the elements of $\mathbf{m}[p]$ *tokens*. The dynamics of a SN is defined by the *firing rule*. We assume that missing arcs are annotated by *empty* bag-functions.

An instance (t, b) *has concession* in a marking $\mathbf{m}$ if and only if:

– $\forall p \in P$: $\mathrm{I}[p, t](b) \leq \mathbf{m}[p]$
– $\forall p \in P, x \in \mathrm{H}[p, t](b)$: $\mathrm{H}[p, t](b)(x) > \mathbf{m}[p](x)$

An instance (t, b) is *enabled* in $\mathbf{m}$ if and only if (t, b) has concession in $\mathbf{m}$ and there is no higher priority transition's instance having concession in $\mathbf{m}$. Assuming the aforementioned (initial) marking for place `Sender` (i.e., a single token $\langle s_1, data\rangle$), the instance ($n = s_1$, $l = data$) of `broadcast` is enabled. If enabled, (t, b) may *fire*, leading to a marking $\mathbf{m}'$ formally defined as:

$$\forall p : \mathbf{m}'[p] = \mathbf{m}[p] - \mathrm{I}[p, t](b) + \mathrm{O}[p, t](b)$$

$\mathbf{m}'$ is said *reachable* from $\mathbf{m}$ through (t, b), and this is denoted $\mathbf{m}[t, b\rangle\mathbf{m}'$. The firing of (`broadcast`, $n = s_1$, $l = data$), according to the arc-function semantics illustrated in the next section, withdraws the token $\langle s_1, data\rangle$ from the input place `Sender` and puts the (type-set) bag of tokens $\sum_{x \in \mathrm{N}, x \neq s_1} 1 \cdot \langle s_1, x, data\rangle$ into output place `EnRoute`, representing a broadcast data-message sent by node s_1.

A marking $\mathbf{m}$ is said *vanishing* if there are some immediate transition instances enabled in $\mathbf{m}$, *tangible* otherwise. A SN *model* is a SN with a *tangible initial marking* $\mathbf{m}_0$. Assuming that there are no infinite sequences of immediate transition instances, it is possible to define the *tangible reachability graph* (TRG) of a SN model, an edge-labelled, directed multi-graph (V, E) whose nodes are tangible markings: $\mathbf{m}_0 \in V$; if $\mathbf{m} \in V$ and there exists a (possibly *empty*) sequence $\{(t_i, b_i), \pi(t_i) > 0\}$, $i : 1 \ldots \mathrm{n} \in \mathbb{N}$, such that $\mathbf{m}[t, b\rangle\mathbf{m}_1[t_1, b_1\rangle \ldots \mathbf{m}_\mathrm{n}[t_\mathrm{n}, b_\mathrm{n}\rangle\mathbf{m}'$, with $\mathbf{m}'$ tangible, $\mathbf{m}' \neq \mathbf{m}$, then $\mathbf{m}' \in V$ and $\mathbf{m} \xrightarrow{t,b} \mathbf{m}' \in E$.

2.3.3 Guards and Arc-functions Syntax

We use a simple convention for color-classes and variables. Color classes are denoted by single capital letters in normal font, e.g., V. The i-th static subclass of V, if any, is denoted by V_i. The set $Var(t)$, $t \in T$, includes all and only the variable symbols occurring in $g(t)$ or in any function $\mathrm{I}[p, t]$, $\mathrm{O}[p, t]$, $\mathrm{H}[p, t]$, $\forall p \in P$. A variable is denoted by a single lowercase letter, which implicitly refers to the variable's type, e.g., v_i is a class-V variable. The number of V-variables in $Var(t)$ coincides with the repetitions e^{V} of V in $\mathcal{D}(t)$: a subscript i ranging in $1 \ldots e^{\mathrm{V}}$ is used to differentiate variables of the same class. It may be omitted if $e^{\mathrm{V}} = 1$.

Projections The adopted convention outlines the transition variable semantics: $v_i \in Var(t)$ is a function $\mathcal{D}(t) \rightarrow Bag[\mathrm{V}]$ that maps a color-tuple b to the i-th occurrence of color V in b (formally, a singleton bag).

Guards A transition *guard* $g(t) : \mathcal{D}(t) \rightarrow \{true, false\}$ is defined in terms of *basic predicate*s and the usual logical connectives. Referring to a generic class V, a basic predicate is any of the following:

- $v_1 = (\neq)\ v_2\ true$ (for a given $b \in \mathcal{D}(t)$) when
  $v_1(b) = (\neq)\ v_2(b)$
- $v_i \in (\notin)\ \mathrm{V}_j\ true$ when $v_1(b)$ belongs (does not belong) to subclass V_j
- $d(v_1) = (\neq) d(v_2)\ true$ when $v_1(b)$, $v_2(b)$ belong to the same (different) subclass(es).[8]

The 2nd and 3rd predicates are admitted if the class V is partitioned. If V is ordered, then in the first type of predicate projections may be suffixed by $++$, $--$, denoting the $\mathrm{mod}_{|\mathrm{V}|}$ successor/predecessor of selected color.

For example, the guard of transition `lostMsg` in Fig. 1, where $\mathcal{D}(\texttt{lostMsg}) = \mathrm{N}^2 \times \mathrm{L}$, indicates that only messages that are sent from a sub-network to the other may get lost.

Arc-function syntax SN *arc-functions* are built of *class-functions*. We consider any arc linking a place p to a transition t and assume that color class V occurs in $\mathcal{D}(p)$.

A class-V function f_i is a map $\mathcal{D}(t) \rightarrow Bag[\mathrm{V}]$, which is defined as a linear combination of (type-set) *elementary functions*:

$$f_i = \sum_h \alpha_h.e_h,\ \alpha_h \in \mathbb{Z}, \tag{5}$$

where e_h may be any of $\{v_j, \mathrm{V}_q, \mathrm{V}\}$ (symbol *All* may be used in place of V):

- v_j is a *projection*, possibly suffixed (if V is ordered) by $++$, $--$
- V_q (admitted if V is partitioned) and V (*All*) are *constant* functions mapping to $\sum_{x \in \mathrm{V}_q} 1 \cdot x$ and $\sum_{x \in \mathrm{V}} 1 \cdot x$, respectively.

The class-N function $All{-}n$ appearing on the output arc from `broadcast` to `EnRoute` (Fig. 1), e.g., maps a color pair $\langle c_1, c_2 \rangle \in \mathrm{N} \times \mathrm{L} = \mathcal{D}(\texttt{broadcast})$ to $\sum_{x \in \mathrm{N}, x \neq c_1} 1 \cdot x$, which represents the whole set of network nodes but c_1.

An arc-function $F[p, t] : \mathcal{D}(t) \rightarrow Bag[\mathcal{D}(p)]$ is in turn defined as a linear combination:

$$F[p, t] = \sum_k \lambda_k.T_k[g_k],\ \lambda_k \in \mathbb{Z}, \tag{6}$$

where T_k is a Cartesian product $\langle f_1, \ldots, f_\mathrm{n} \rangle$ of class-functions[9] (Section 2.1, Eqs. 2) and g_k is an (optional) guard defined on $\mathcal{D}(t)$, with the same syntax as transition guards: $T_k[g_k](b) = T_k(b)$ if $g_k(b) = true$, otherwise $T_k[g_k](b) = \emptyset$, $\forall b \in \mathcal{D}(t)$.

Scalars in Eqs. 5 and 6 must ensure that linear combinations are well-defined.

The arc-function $\mathrm{O}[\texttt{EnRoute}, \texttt{broadcast}] = \langle n, All - n, l \rangle$ (Fig. 1), e.g., when evaluates on $\langle c_1, c_2 \rangle \in \mathrm{N} \times \mathrm{L} = \mathcal{D}(\texttt{broadcast})$ results in $\langle c_1, \sum_{x \in \mathrm{N}, x \neq c_1} 1 \cdot x, c_2 \rangle$, a bag Cartesian product which corresponds to all triplets with c_1 as 1st element, a node other than c_1 as 2nd element, and c_2 as 3rd element. That is, a type-c_2 broadcast message sent by node c_1.

A worthwhile property of SN arc-functions is that any $F[p, t]$ can be equivalently expressed as $\sum_i \lambda_i T_i[g_i]$,

where each term $T_i[g_i]$ is:

[8] This syntax is not currently supported by the GREATSPN GUI.

[9] The color-class of each f_i has to match $\mathcal{D}(p)$.

- type-set
- pairwise-disjoint
- constant-size, i.e, $\exists k \in \mathbb{N}^+ \ \forall b \ g_i(b) = true \Rightarrow |T_i(b)| = k$

For example, $2 \cdot \langle All - v_1, v_2\rangle[v_1 \neq v_2] + 1 \cdot \langle v_2, v_2\rangle \ : \ \mathrm{V}^{\mathrm{k}} \rightarrow \mathrm{V}^2, \ \mathrm{k} > 1 \equiv 2 \cdot \langle All - v_1 - v_2, v_2\rangle[v_1 \neq v_2] + 3 \cdot \langle v_2, v_2\rangle[v_1 \neq v_2] + 1 \cdot \langle v_2, v_2\rangle[v_1 = v_2]$. The size of $\langle All - v_1 - v_2, v_2\rangle[v_1 \neq v_2]$ is $|\mathrm{V}| - 2$, the other tuples are size-one.

We may therefore assume that arc functions are linear combinations of type-set, constant-size (and, if needed, pair-wise disjoint) function-tuples.

3 The self-healing manufacturing system case study

In this section, we introduce a self-healing (or self-repairing) manufacturing system example to put into place the major concepts of our approach. Here adaptation procedures run over a distributed infrastructure in order to reconfigure the usage of the available resources depending on the current condition of the system itself and the environment. To achieve this goal, the system defines primitive actions to be applied in production, such as connection/inhibition of the available/faulty production lines and migration of the raw products among available machines of the production system.

In our running example, we consider a Manufacturing System (MS) composed of two production lines, that refine a number of raw pieces in order to compose the final product. Typically, the reconfiguration process consists in deciding whether to inhibit faulty components and/or migrate raw products towards available resources of the system, depending on a set of metrics (either simple or derived). For the sake of simplicity, we confine our example to two metrics: the *status* of the available production lines and the *workload* in terms of assigned raw products. The value associated to these metrics will be used to plan and execute the adaptation. In particular, the system must be endowed with the ability of recognizing the status of the production lines and change accordingly the current availability of resources and the current workload. As described by Dicesare et al. (1993), Meng (2010), and Capra (2016), this scenario is very common in smart MS, where reorganizing or changing components must be taken into account in the system design in order to achieve robustness, flexibility and resiliency in the production environment.

Symmetrical fault schema Fig. 2 shows a P/T net modeling the MS with a symmetrical fault schema (i.e., SMS). Two production lines are represented by the subnets { $\mathtt{pl}_1$, $\mathtt{tl}_1$, $\mathtt{pw}_1$ } and { $\mathtt{pl}_2$, $\mathtt{tl}_2$, $\mathtt{pw}_2$ }, respectively. The two lines refine raw pieces of the same type, represented by tokens inside the two corresponding subnets. Pairs of refined pieces are taken from both lines, then assembled to get the final product (represented by the firing of $\mathtt{ta}_1$, i.e., the `assembler` component). The `loader` component (transition $\mathtt{tlo}_1$) initializes the whole process by picking up two raw pieces a time from the `storage` component (place $\mathtt{pin}_1$) initially holding N pieces, and putting them onto the lines. After the expected number of final items is obtained, the system starts again (transition $\mathtt{trs}_1$). The model contains also the specification of a faulty behavior. In fact, a product line is periodically subject to failures (represented by $\mathtt{tfa}_1$ and $\mathtt{tfa}_2$). Faults can occur and block the lines. Faults are modeled by means of tokens inside the place $\mathtt{pb}_1$ (i.e., faults in the first line) or inside pb_2 (i.e., faults in the second line). We here assume that the probability of having failures on both the lines at the same time is negligible.

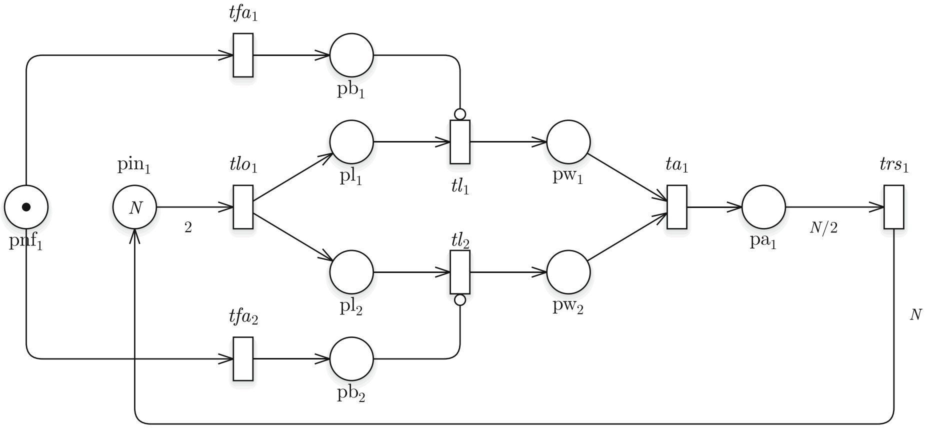


Fig. 2 P/T net modeling the nominal behavior of the SMS

For this reason, the transitions `tfa`$_1$ and `tfa`$_2$ (representing the occurrence of a failure in the first and second production line, respectively) are in structural conflict. This setting can be easily generalized to the case where failures may arbitrarily occur at any time on any production line. Nevertheless, this choice simplifies the description of the adaptation procedures reported in the next sections.

Asymmetrical fault schema. The Asymmetrical version of the previous model (i.e., AMS) can be obtained by removing either the transition `tfa`$_1$ or `tfa`$_2$. In this case, a single product line is subject to failures. As the previous model, the occurrence of a failure causes the line to be blocked.

Indeed, blocked lines hinder the liveness of the whole production process. Thus, a common adaptation scenario for both the SMS and the AMS is the reconfiguration of the system to handle occurrences of faults. In fact, the production should continue without shutting down the system using the available resources (i.e., the remaining production lines). The faulty lines must be inhibited or detached from the system, and the behavior of both the `loader` and the `assembler` must be changed accordingly. Namely, the `loader` must put two raw pieces a time on the available line, and the `assembler` must take pairs of refined pieces from the working line. As soon as the failing component is repaired, the system should go back to its nominal configuration.

Although P/T nets represent an expressive model of concurrency and distribution, the adaptation scenario reported above (involving structural changes of the model) cannot be specified in a natural way. In the next section, we describe how our modeling framework can be used to overcome this limitation by introducing the self-adaptation process with clear separation of concerns.

4 Preview of the approach

According to de Lemos et al. (2013), we adopt the general terms *managed* (or *base-level*) and *managing* (or *reflective*) subsystems to denote the two main parts of an adaptable/evolvable system.

The environment refers to the external world, such as physical and/or software components, in which the system runs/interacts, and where the effects of the system (in its nominal and adapted behaviors) will be observed. The managed subsystem provides support for monitoring and executing adaptations controlled by the managing subsystem. This latter component contains the adaptation logic that deals with one or more concerns. In our framework, the adaptation logic is viewed as a set of *adaptation procedure*s realizing multiple (concurrent) feedback control loops, as introduced by Brun et al. (2009). We assume the target system (both managed and managing components) is distributed and we assume *decentralization* of the control decisions.

In particular, we consider decentralization at the level of the adaptation procedures, as introduced by de Lemos et al. (2013). Figure 3 shows an abstract overview of our modeling framework. In this section we introduce the rationale of the major components that we further detail in Section 5.

The *base-level* contains the definition of the managed subsystem as a P/T system. It describes the system's nominal behavior and its interaction with the environment. For instance, the SMS model in Fig. 2 specifies the nominal behavior of our case study. Such a model represents the base-level layer of a self-adaptive manufacturing system.

The base-level is then endowed with self-adaptation capability.

These additional features are specified in the *high-level* layer which implements the system's adaptation logics. In our modeling framework, the components inspecting and possibly modifying the base-level are formalized as SN models. The structure of the higher-level is shown in Fig. 3. The major components are as follows:

- *emulator*;
- *net-interface*;
- atomic *transformation rules*;
- *evolutionary API*;

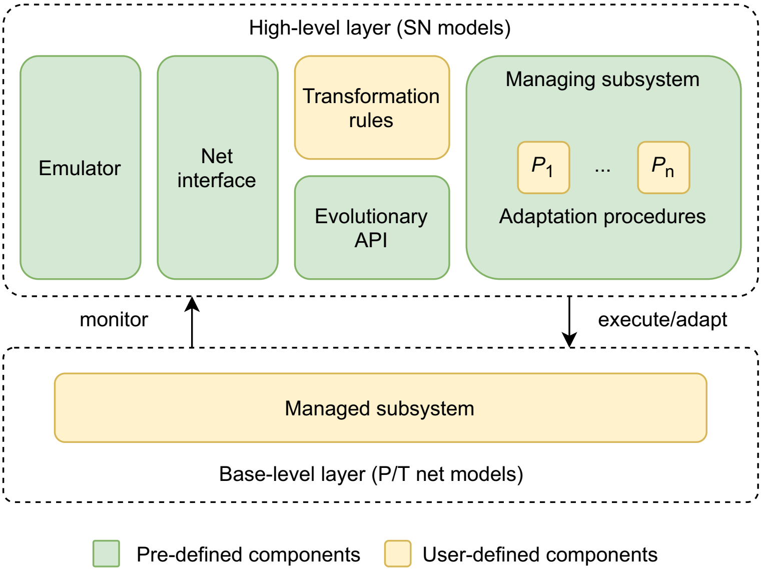


Fig. 3 Preview of the modeling framework

- *managing subsystem* as a number of *adaptation procedures*.

The *emulator* is a built-in SN model provided by our framework. The role of the emulator is to *encode* a given P/T net and *emulate* its dynamics. The target base-level (a P/T system) is encoded as a marking of the emulator, so that it can be manipulated by transitions linked to the emulator net through a well-defined interface. The base-level manipulation may cause base-level's structure/state be consistently changed, and may be concurrent with emulation. The modeler can specify and validate atomic *transformation rules* to be applied to the encoded P/T net. Transitions are provided through SN models. A symbolic structural calculus for SN is used to (automatically) check the soundness of transformation rules, and safely link them (through arc functions) to the rest of the model. The *evolutionary API* provides a basic (yet complete) collection of pre-defined primitive *transformation*s to perform *introspection*/*intercession* on the base-level. For instance, the API can provide the current configuration of the encoded system, add/remove P/T nodes and edges, and change its marking. Each primitive is defined by a SN subnet that modifies the encoded net consistently and atomically. The *net-interface* plays a crucial role, providing an interface between user-defined transformations/API primitives and the emulator. It manages, in particular, the consistent update of essential data-structures (SN places) which are used in the emulation algorithm to improve efficiency.

The *managing subsystem* contains a collection of *adaptation procedure*s, specified as disjoint SNs. Each procedure manages an adaptation concern, and implements a feedback loop monitoring the base-level subsystem. In our selected case study, the following adaptation concerns are treated:

(i) *fault tolerance* – continue producing in presence of a faulty line, which is temporarily put off-line, without shutting down the system;
(ii) *load balancing* – bring raw pieces to all the available production lines, to optimize resource usage.

Adaptation procedures are specified independently, and are activated in specific conditions evaluated on the base-level encoding. Procedures are concurrent, also with the base-level dynamics, and operate different adaptation plans. In this sense, we do not assume the existence of a centralized controller. The modeling framework guarantees a theoretically consistent evolution of the system, whereas logical correctness is formally checked by using consolidated analysis techniques for SNs.

Procedures may use predefined, primitive API operations representing a base set of *sensors* and *actuators*. In addition, they may also integrate user-defined transformation rules, that enhance model flexibility. For instance, the procedure (*i*) *fault tolerance*, directly accesses the emulator's evolutionary interface (a set of SN places encoding the base-level) to recognize the presence of a token either in $\mathtt{pb}_1$ or $\mathtt{pb}_2$ (a failure occurrence). When a failure occurs, it carries out a number of changes to the base-level by using write API primitives. For instance, residual raw pieces on the faulty line are moved to the working one. Then, the loader and assembler components are adapted to ensure system operation even in presence of a failure.

Our modeling framework also allows temporal aspects to be specified. In this case, the modeler uses the native stochastic extension of SNs (i.e., SSNs) to enrich the system specification. Namely, transitions representing observable events can be associated with *rate*s characterizing exponential firing delays.

5 The emulation framework

The emulator reproduces the *interleaving* semantics of a P/T system encoded as a SN colored marking. In this section, we first introduce the notion of *net encoding* in Section 5.1, and then we formally describe the behavior and the structural properties of the emulator in Section 5.2. We introduce the notion of user-defined *transformation rules* in Section 5.3 and we describe how to link them to the emulator through the *net-interface* in Section 5.4. We also describe the set of pre-defined primitives that compose the *evolutionary API* in Section 5.5.

5.1 P/T nets encoding

As anticipated, our approach is based on encoding a P/T net as the marking of specific places of the emulator SN. These places compose the *evolutionary interface*, which is defined as follows:

$$\mathbf{EI} = \{\mathtt{IN}, \mathtt{H}, \mathtt{O\_I}, \mathtt{I\_O}, \mathtt{MARK}\} \tag{7}$$

Intuitively, the **EI** is suitably linked to both the emulator model (that cause the encoded P/T to be executed) and additional SNs implementing net-transformation rules (comparable to the rules of graph transformation systems).

The emulator's color structure builds on two color-classes P, T, holding descriptors for the nodes of a P/T net, and on color domain $Arc = \mathrm{P} \times \mathrm{T}$, whose color-tuples describe the edges of a P/T net. For the sake of simplicity, we will use the same symbols for colors and corresponding P/T nodes, letting the context disambiguate their role. We assume that classes P and T are either circularly ordered or partitioned for modeling reasons, as detailed in the following.

Let $(N, m) = (P, T, I, O, H, m)$ be a P/T system. The SN places IN, H, O_I, I_O encode the graph structure N, whereas place MARK encodes m. We assume $|P| \leq |\mathrm{P}|$, $|T| \leq |\mathrm{T}|$ or, according to the adopted convention, $P \subseteq \mathrm{P}$, $T \subseteq \mathrm{T}$.

Definition 1 (encoding of (N, m)) Let $(N, m) = (P, T, I, O, H, m)$ be a P/T system. An SN marking $\mathbf{m}$ is an encoding of (N, m) if and only if:

$$\mathbf{m}[\mathtt{IN}] = I \qquad \mathbf{m}[\mathtt{H}] = H \qquad \mathbf{m}[\mathtt{O\_I}] = O - I \qquad \mathbf{m}[\mathtt{I\_O}] = I - O \qquad \mathbf{m}[\mathtt{MARK}] = m.$$

The encoding mechanism allows the firing of P/T transitions to be emulated efficiently. The explicit representation of bag differences $I - O$ and $O - I$ reduces the complexity of the computation of transition firing effects. In the following, we use the notation $\mathbf{m}_N$ to indicate a SN marking which encodes a given P/T net.

The other way around, the conditions ensuring that a SN marking encodes a P/T system are formally expressed by:

Definition 2 (P/T-encoding marking) $\mathbf{m}$ is a P/T encoding if and only if

1. $\mathbf{m}[\mathtt{O\_I}] + \mathbf{m}[\mathtt{IN}] + \mathbf{m}[\mathtt{H}] \neq \emptyset$
2. $\mathbf{m}[\mathtt{I\_O}] \leq \mathbf{m}[\mathtt{IN}] \wedge \ \mathbf{m}[\mathtt{O\_I}] \cap \mathbf{m}[\mathtt{I\_O}] = \emptyset$
3. $\overline{\mathbf{m}[\mathtt{MARK}]} \subseteq \overline{p_1 \circ (\mathbf{m}[\mathtt{IN}] + \mathbf{m}[\mathtt{O\_I}] + \mathbf{m}[\mathtt{H}])}$

The 1st condition is related to non-emptiness of a P/T net. The 2nd condition to the meaning of bag differences $I - O$ and $O - I$, and to the fact that output edges of a P/T net are implicitly represented. The third condition ensures the absence of isolated places: p_1 is the projection P × T → P.

The absence of isolated nodes is relevant in a context where the structure/state of the managed subsystem may change due to adaptation procedures. In this setting, spurious transformations may lead a system to an inconsistent state. The implicit representation of P/T nodes by means of edges avoids any trickiness due to possible dangling edges, as it happens in graph transformation approaches.

The P/T net corresponding to a graph-encoding **m** is defined as follows.

Definition 3 (encoded P/T system) Let **m** be a P/T encoding (Def. 2). The encoded P/T system is (P, T, I, O, H, m), where P, T are implicitly defined by I, O, H and

$$I = \mathbf{m}[\mathtt{IN}] \qquad H = \mathbf{m}[\mathtt{H}] \qquad O = \mathbf{m}[\mathtt{O\_I}] + \mathbf{m}[\mathtt{IN}] - \mathbf{m}[\mathtt{I\_O}] \qquad m = \mathbf{m}[\mathtt{MARK}]$$

Property 1 Denoting with $\mathbf{m}(N, m)$ the encoding of a P/T system and with $\mathcal{N}(\mathbf{m})$ the P/T system encoded by a P/T encoding **m**, it holds $\mathcal{N}(\mathbf{m}(N, m)) = (N, m)$.

As an example, the encoding of the *AMS* model introduced in Section 3 is as follows.

$$\begin{aligned}
\mathbf{m}[\mathtt{IN}] &= \langle \mathtt{pnf}_1, \mathtt{tfa}_1 \rangle + \langle \mathtt{pnf}_1, \mathtt{tfa}_2 \rangle + 2 \cdot \langle \mathtt{pin}_1, \mathtt{tlo}_1 \rangle + \langle \mathtt{pl}_1, \mathtt{tli}_1 \rangle \\
&\quad + \langle \mathtt{pl}_2, \mathtt{tli}_2 \rangle + \langle \mathtt{pw}_1, \mathtt{ta}_1 x \rangle + \langle \mathtt{pw}_2, \mathtt{ta}_1 \rangle + \ N/2 \cdot \langle \mathtt{pa}_1, \mathtt{trs}_1 \rangle \\
\mathbf{m}[\mathtt{H}] &= \langle \mathtt{pb}_1, \mathtt{tli}_1 \rangle + \langle \mathtt{pb}_2, \mathtt{tli}_2 \rangle \\
\mathbf{m}[\mathtt{O\_I}] &= \langle \mathtt{pl}_1, \mathtt{tlo}_1 \rangle + \langle \mathtt{pl2}_1, \mathtt{tlo}_1 \rangle + \langle \mathtt{pw}_1, \mathtt{tli}_1 \rangle + \langle \mathtt{pw}_2, \mathtt{tli}_2 \rangle + \langle \mathtt{pa}_1, \mathtt{ta}_1 \rangle \\
&\quad + N \cdot \langle \mathtt{pin}_1, \mathtt{trs}_1 \rangle + \langle \mathtt{pb}_1, \mathtt{tfa}_1 \rangle + \langle \mathtt{pb}_2, \mathtt{tfa}_2 \rangle \\
\mathbf{m}[\mathtt{I}_O] &= \mathbf{m}[\mathtt{IN}] \\
\mathbf{m}[\mathtt{MARK}] &= N \cdot \mathtt{pin}_1 + \mathtt{pnf}_1
\end{aligned}$$

5.2 The emulator

For the sake of readability, we introduce the emulator component through the pseudocode in Algorithm 1. The complete SN specification is available in Appendix A.

The rationale of the main operations performed by the emulator can be summarized as follows. A reachable *tangible* marking of the SN matches a reachable marking of the encoded P/T net (N, m), an enabled instance of the only observable transition PT_fire with $\mathcal{D}(\mathtt{PT_fire}) = $ T, matches an enabled P/T transition. When a firing occurs, an instance $(\mathtt{PT\_fire}, t = t_k)$ triggers a sequence σ of *immediate* transition instances emulating the firing of t_k, in accordance with the *atomic* semantics of Petri nets. The main steps executed in the sequence σ are denoted by steps *i*, *ii*, and *iii* in the pseudocode (and the SN model in Appendix A). The description of these steps follows.

i This step is triggered by a token in neutral place beginTestEnab. All and only P/T transitions in the place checkList are marked as "to be checked". These transitions are tested for enabling, by moving tokens from checkList to toTest and by considering first *input* places then *inhibitor* places (once a time). At the end of this process the place checkList is emptied, and enabList is filled up. The step *i* ends by firing the transition endTestEnab.

ii This step starts with the firing of a color-instance of transition `PT_fire`. The effect is to non-deterministically chose a P/T transition t (according to the semantics of P/T nets) from place `enabList`. Thus, the marking of place `MARK` is modified, according to the firing rule of P/T systems. Namely, for each element in $(I - O)(p, t)$, $m(p)$ is decremented and for each element in $(O - I)(p, t)$, $m(p)$ is incremented. This step is triggered by a token in neutral place `beginFiring`.

iii This step updates the marking of two places holding the transitions that were enabled before the firing of t (place `enabList`) and those whose enabling must be checked upon it (place `checkList`), respectively, by taking account of the (asymmetric) *Structural Conflict* (SC) and *Structural Causal Connection* (SCC) relations between P/T transitions, formalized at lines 24 and 30 of Algorithm 1, respectively: $t\ SC\ t_2$ if and only if t *may* disable t_2 in a marking, whereas $t\ SCC\ t_2$ if and only if t *may* enable t_2. The use of these two auxiliary places may greatly enhance the efficiency of the whole emulation process.

Algorithm 1 The SN emulator behavior represented as pseudocode. Utility functions are reported in Alg. 2. Legend: $^\bullet t$ pre-set of t, $^\circ t$ inhibitor set of t.

```
 1: function EMULATE($P, T, I, O, H, m$)
 2:     $enabList = \emptyset$
 3:     $checkList = T$
 4:     while $True$ do
 5:         for all $t \in checkList$ do                                  ▷ step i
 6:             $checkList = checkList \setminus \{t\}$
 7:             $enab_t$ = CHECKINARCS($t$)
 8:             if $enab_t$ then
 9:                 $enab_t$ = CHECKHARCS($t$)
10:             end if
11:             if $enab_t$ then
12:                 $enabList = enabList \cup \{t\}$
13:             end if
14:         end for
15:         if $enabList = \emptyset$ then
16:             exit
17:         else
18:             let $t \in enabList$                          ▷ P/T transition to be fired
19:             $enabList = enabList \setminus \{t\}$
20:             $checkList = checkList \cup \{t\}$
21:             CHANGEPRESET()                                            ▷ step ii
22:             CHANGEPOSTSET()
23:             for all $t_2 \in enabList$ do                             ▷ step iii
24:                 if $(\exists p \in {}^\bullet t_2 : (I - O)(p, t) > 0) \ \vee \exists p \in {}^\circ t_2 : (O - I)(p, t) > 0)$ then
25:                     $enabList = enabList \setminus \{t_2\}$
26:                     $checkList = checkList \cup \{t_2\}$
27:                 end if
28:             end for
29:             for all $t_2 \in T \setminus (enabList \cup checkList)$ do
30:                 if $(\exists p \in {}^\bullet t_2 : (O - I)(p, t) > 0) \ \vee (\exists p \in {}^\circ t_2 : (I - O)(p, t) > 0)$ then
31:                     $checkList = checkList \cup \{t_2\}$
32:                 end if
33:             end for
34:         end if
35:     end while
36: end function
```

Algorithm 2 Functions used by Alg. 1.

```
function CHECKINARCS($t$)
    for all $p \in {}^{\bullet}t$ do
        if $I(p,t) > m(p)$ then
            return $False$
        end if
    end for
    return $True$
end function
function CHECKHARCS($t$)
    for all $p \in {}^{\circ}t$ do
        if $H(p,t) \leq m(p)$ then
            return $False$
        end if
    end for
    return $True$
end function
function CHANGEPRESET()
    for all $p : (I - O)(p,t) > 0$ do
        $count = (I - O)(p,t)$
        do
            $m(p)--$
        while $count-- \neq 0$
    end for
end function
function CHANGEPOSTSET()
    for all $p : (I - O)(p,t) > 0$ do
        $count = (I - O)(p,t)$
        do
            $m(p)--$
        while $count-- \neq 0$
    end for
end function
```

The behavior of the emulator SN which is mechanically translated from Algorithm 1 is formalized as follows.

Property 2 Let (N, m) be a P/T system, $E(m)$ the set of transitions enabled in m and $\mathbf{m}$ an emulator's *tangible* marking encoding (N, m) (Definition 1) such that:

1. $\mathbf{m}[\texttt{enabList}] = \sum_{tr \in E(m)} 1 \cdot tr$,
   $\mathbf{m}[\texttt{toTest}] = nil$,[10] $\mathbf{m}[\texttt{beginFiring}] = 1$
2. $\mathbf{m}[p] = \emptyset$ (0), for every other place p

Then $\mathbf{m} \xrightarrow{\texttt{PT\_fire},tr_k} \mathbf{m}'$, for any $tr_k \in$ T, if and only if $m[tr_k\rangle m'$, where $\mathbf{m}'$ is the tangible marking encoding (N, m') such that $\mathbf{m}'[\texttt{enabList}] = \sum_{tr \in E(m')} 1 \cdot tr$ and $\mathbf{m}'[p] = \mathbf{m}[p]$, for any other place p.

As a direct corollary of Property 2, we have:

[10] nil represents a kind of default class-T color.

Corollary 1 *The tangible reachability graph of the emulator whose initial marking* $\mathbf{m}_0$ *encodes a P/T system* (N, m_0) *and matches conditions 1,2 of Property 2 is isomorphic to the reachability graph of* (N, m_0).[11]

The emulation process must coherently resume after any structural change to the encoded system. Property 3 defines a state (vanishing marking) from which the emulation of a P/T system may coherently restart. The rationale is that from this state, assuming that all P/T transitions potentially disabled by a structural change were moved from `enabList` to `checkList` then all those potentially enabled (not in `enabList`) were added to `checkList`, the emulator eventually reaches a tangible marking that satisfies Property 2. The net-interface component of the framework (Section 5.4) transparently manages all that.

Property 3 Let (N, m) be a P/T system, $\mathbf{m}$ be an emulator *vanishing* marking encoding (N, m) (Definition 1) such that:

- $\mathbf{m}[\texttt{beginTestEnab}] = 1, \mathbf{m}[\texttt{toTest}] = tr_j$, for any $tr_j \in \mathrm{T}$
- $\mathbf{m}[\texttt{checkList}]$ is type-set
- for every other place p, $p \neq \texttt{enabList}$: $\mathbf{m}[p] = \emptyset$

Then, from $\mathbf{m}$ we *eventually* reach through any vanishing path the *tangible* marking $\mathbf{m}'$ such that:

- $\mathbf{m}'[\texttt{beginFiring}] = 1, \mathbf{m}'[\texttt{toTest}] = nil, \mathbf{m}'[\texttt{checkList}] = \emptyset$
- $\mathbf{m}'[\texttt{enabList}] = \mathbf{m}[\texttt{enabList}] + (\sum_{tr \in E(m)} 1 \cdot tr) \cap \mathbf{m}[\texttt{checkList}]$
- for every other place p, $\mathbf{m}'[p] = \mathbf{m}[p]$.

5.3 Transformation rules

We first introduce a general transformation rule of the encoded system, as a user-defined SN transition that is directly connected to the emulator's interface **EI** (7). Then we present an API of base transformation primitives. Transformation rules/primitives are transparently linked to the emulator through a net interface.

We follow a structural approach, by defining parametric (i.e., not depending on the encoded system) *structural* conditions ensuring that transformations are consistent, according to Def. 4, and the emulation of the encoded system is correct also in the face of changes to its state/topology. All these conditions (stated in a series of lemmas) can be automatically checked using the SN *calculus* introduced by Capra et al. (2005, 2015) and implemented on the tool SNEXPRESSION.[12] In such a calculus, structural conditions are formalized as symbolic expressions that are directly derived from SN arc-functions and manipulated algebraically.

Definition 4 (Valid P/T transformation) A SN transition t linked to the emulator's evolutionary interface **EI** is a valid P/T transformation if and only if, for any P/T encoding $\mathbf{m}$, $\forall b \in \mathcal{D}(t)$: $\mathbf{m}[t, b\rangle\mathbf{m}' \Rightarrow \mathbf{m}'$ is a P/T encoding.

[11] $E(m_0)$ can be automatically computed, instead of being pre-calculated, by setting a transient initial (tangible) marking with a token in neutral place `startUp`, $\langle All \rangle$ in `checkList`, and with places `beginFiring` and `enabList` empty. From such a marking we eventually reach $\mathbf{m}_0$

[12] Publicly available at: www.di.unito.it/~depierro/SNex.

The SN calculus provides support for the computation of the following structural properties: *conflict*, *causal connection*, *mutual exclusion*, and some kinds of coloured *semi-flows*. It builds on the ability to solve symbolic formulae that are defined in terms of a language $\mathcal{L}$ and a base set of functional operators, under which $\mathcal{L}$ is closed: difference, intersection, composition, transpose, and bag support. The terms of $\mathcal{L}$ are SN arc functions possibly prefixed by guards (called *filters*) on function co-domains.

Let $f : A \to Bag[B]$, then $[g]f : A \to Bag[B]$ is:

$$[g]f(a)(x) = f(a)(x), \text{ if } g(x) = \text{true}, \text{ otherwise } [g]f(a)(x) = 0, \forall a \in A, x \in B \quad (8)$$

A helpful operator that can be derived from the available ones is the function *restriction*. Let $F : A \to Bag[B]$, $G : A \to 2^B$, then $F_{[G]} : A \to Bag[B]$ is defined as:

$$F_{[G]}(a)(b) = F(a)(b) \text{ if } b \in G(a), \text{ otherwise } F_{[G]}(a)(b) = 0, \forall a, b \quad (9)$$

The SN calculus operates as a rewriting system. Any expression (formula) e is reduced to a *normal form* $e' \in \mathcal{L}$, where tuples may contain intersections of class-functions and be prefixed by filters. A term's normalization is denoted $e \to e'$.

As described by Capra et al. (2015), equivalence between expressions can be syntactically proven, thanks to the following base property of the calculus:

$$e \equiv \emptyset \;\Leftrightarrow e \to \emptyset \quad (10)$$

The ability to syntactically check equivalences is exploited in relational operators. Let e, e' be bag-expressions of the same arity:

$$e \leq e' \Leftrightarrow e - e' \equiv \emptyset \quad \overline{e} \subseteq \overline{e'} \Leftrightarrow \overline{e} - \overline{e'} \equiv \emptyset \quad (11)$$

You can also syntactically verify whether an expression never evaluates to empty, i.e.:

$$\forall d \in \mathcal{D}(e)\; e(d) \neq \emptyset \quad \text{denoted as} \quad \mathbf{G}\, e \not\equiv \emptyset \quad (12)$$

If e is suffixed by a guard g, then Eq. 12 only applies to those d satisfying g.

Two basic place *invariant* properties used to verify the validity of P/T transformations follow.

Definition 5 (Mutually-exclusive places) Let p, p' be any two different SN places such that $\mathcal{D}(p) = \mathcal{D}(p')$. p, p' are mutually-exclusive ($p\ ME\ p'$) if and only if $\mathbf{m}[p] \cap \mathbf{m}[p'] = \emptyset$, for every reachable marking $\mathbf{m}$.

Definition 6 (Color-safe place) $p \in P$ is color-safe if and only if $\mathbf{m}[p]$ is type-set, for every reachable marking $\mathbf{m}$.

In the sequel, we assume that a (non-trivial) transition guard $g(t)$ is implicitly appended to each adjacent arc-function. That is, $F(p,t)$ becomes $F(p,t)[g(t)]$.

A mostly structural characterization of SN transitions preserving place mutual-exclusion is formalized as follows.

Some auxiliary formulae defined/described in Table 1 are used. $All_{D,D'}$ denotes the function-tuple $D \to Bag[D']$ uniquely composed of All class-functions.

Lemma 1 (P mutual-exclusion preservation) *Let p, p' be any two different SN places such that $\mathcal{D}(p) = \mathcal{D}(p')$, $t \in T$. If $\mathbf{m}[p] \cap \mathbf{m}[p'] = \emptyset$ and, given A, B, C defined below, $A \wedge (B \vee C) \equiv true$ and the other way around, by swapping p and p', then:*

$$\boldsymbol{m}[t, b\rangle\boldsymbol{m}' \Rightarrow \boldsymbol{m}'[p] \cap \boldsymbol{m}'[p'] = \emptyset, \forall b \in \mathcal{D}(t).$$

Table 1 Auxiliary symbolic expressions

Symbol	Definition	Description
$\mathcal{D}(t) \rightarrow Bag[\mathcal{D}(p)]$		
$\mathrm{W}^{+}[p,t]$	O[p,t]−I[p,t]	Multiset of colors actually put in place p by an instance of t.
$\mathrm{W}^{-}[p,t]$	I[p,t]−O[p,t]	Multiset of colors actually removed from place p by an instance of t.
$\mathcal{D}(t) \rightarrow 2^{\mathtt{P}}\mathbf{EI}_N = \{\mathtt{IN}, \mathtt{H}, \mathtt{O\_I}\}$		
$AbI(t)$	$\overline{p_1 \circ \sum_{p\in\mathbf{EI}_N} \mathrm{O}[p,t]}$	Set of P-colors representing P/T places that edges (re)inserted by an instance of t are incident to.
$RbI(t)$	$\overline{p_1 \circ \sum_{p\in\mathbf{EI}_N} \mathrm{I}[p,t]} - AbI(t)$	Set of P-colors representing P/T places that only edges potentially erased by an instance of t are incident to.

$$A: \qquad O[p,t] \cap O[p',t] \equiv \emptyset$$

$$B: \qquad \overline{W^{+}[p,t]} \subseteq \overline{H[p',t]} \ \wedge H[p',t]_{[\overline{W^{+}[p,t]}]} - I[p',t] \leq All_{\mathcal{D}(t),\mathcal{D}(p')}$$

$$C: \qquad \overline{W + p, t} \subseteq \overline{I p' t} \ \wedge \boldsymbol{m}[p'] \textit{ is type-set}$$

Proof The condition $A \wedge (B \vee C)$ means that, for any instance $b \in \mathcal{D}(t)$, every color x which is put in place p (i.e., $x \in \mathrm{O}[p,t](b)$) is actually *erased* from p', because A) $x \notin$ p'$t(b)$ and: either B) there is an upper-bound for the occurrences of x in p' due to the inhibitor arc-function, matching the occurrences of x withdrawn from p' (indeed, $\mathrm{H}[p',t](b)(x) - \mathrm{I}[p',t](b)(x) \leq 1$; note that if $\mathrm{I}[p',t]$ is *null*, then $\mathrm{H}[p',t]$ is type-set); or C) x occurs at most once in p' and belongs to $\mathrm{I}[p',t](b)$ (thus it is withdrawn from p', if t is enabled). □

Condition C of Lemma 1 is not entirely structural, however, it extends the applicability of the lemma. We write $p\ SME_t\ p'$ to mean that a transition t meets conditions A and B of Lemma 1, i.e., t preserves mutex of places p, p' *structurally*.

Lemma 2 (Structural characterization of a valid P/T Transformation) *Let* $\boldsymbol{EI}_N = \{\mathtt{IN}, \mathtt{H}, \mathtt{O\_I}\} \subset \boldsymbol{EI}$, $t \in T$.

If the conditions C1-C5 below are satisfied, then t is a valid P/T transformation (Def. 4).

C1 $\forall p \in \boldsymbol{EI}_N\ I[p,t] \equiv \emptyset\ \vee\ \exists p \in \boldsymbol{EI}_N\ \boldsymbol{G}\ O[p,t]\ \not\equiv \emptyset$
C2 $\mathtt{O\_I}\ SME_t\ \mathtt{I\_O}$
C3 $W^{-}[\mathtt{I\_O},t] \geq W^{-}[\mathtt{IN},t] \wedge W^{+}[\mathtt{I\_O},t] \leq W^{+}[\mathtt{IN},t]$
C4 $\overline{O[\mathtt{MARK},t]} - \overline{I[\mathtt{MARK},t]} \subseteq AbI(t)$
C5 $RbI(t) \cap \overline{O[\mathtt{MARK},t]} \equiv \emptyset \ \wedge RbI(t) \subseteq \overline{H[\mathtt{MARK},t]} \ \wedge H[\mathtt{MARK},t]_{[RbI(t)]} - I[\mathtt{MARK},t] \leq \langle All \rangle$

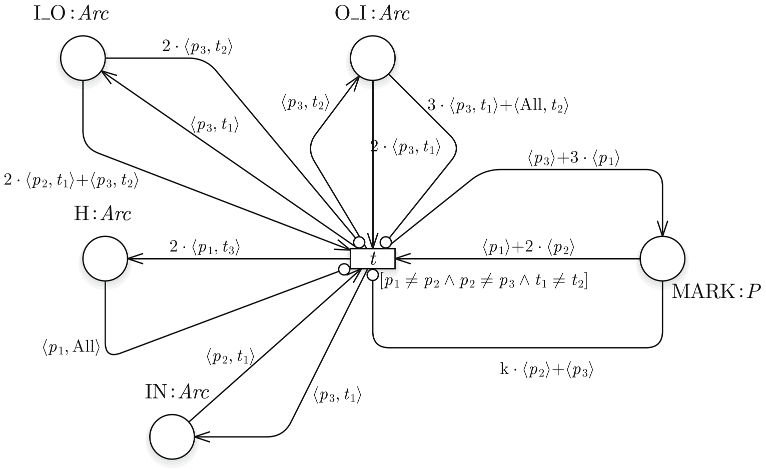
Before the formal proof, let us give an intuitive interpretation for each of the conditions above. We assume to start from a P/T encoding (Def. 2) and to fire any enabled instance of t. $C1$ preserves the non-emptiness of the encoding. $C2$ preserves mutex of places O_I and I_O. As for $C3$, it preserves the inclusion relationship between the marking of places I_O and IN. Finally, $C4$, $C5$ retain the absence of isolated nodes. In particular, $C4$ considers the insertion of new P/T places in the current marking whereas $C5$ deals with the removal of P/T edges (the two operations that may affect that requirement).

Proof Let $\mathbf{m}$ be a P/T encoding, $b \in \mathcal{D}(t)$, and $\mathbf{m}[t, b\rangle\mathbf{m}'$. The condition $C1$ means that no color-tuple (edge) is either removed or added by (t, b).

The condition $C2$ ensures that $\mathbf{m}'[\texttt{O\_I}] \cap \mathbf{m}'[\texttt{I\_O}] = \emptyset$. The condition $C3$ means that each color-tuple (edge) added to place I_O by (t, b) is also added to place IN. In the same way, each color-tuple (edge) withdrawn from place IN is also removed from I_O, thus preserving the relation $\mathbf{m}'[\texttt{I\_O}] \leq \mathbf{m}'[\texttt{IN}]$.

Condition $C4$ can be interpreted as follows: if the instance (t, b) inserts a *potentially new* P-color x in place MARK, then there is some edge (of any type) contextually (re)inserted by (t, b), that x is incident to. However, $C4$ itself is not enough to ensure absence of isolated nodes. In fact, $C5$ considers removal of color-edges by (t, b). It is a conjunctive form, built of three clauses. The set $RbI(t)(b)$ holds P-colors representing places that only edges (of any kind) *potentially erased* by (t, b) are incident to. The first clause of $C5$ ensures that no such color may be added to place MARK. The second clause means that there is an upper bound for each color $x \in RbI(t)(b)$ in SN place MARK, due to a inhibitor arc function (otherwise instance b wouldn't be enabled). The third clause ensures that, for any $x \in RbI(t)(b)$, there cannot be more instances of x in MARK than those which are withdrawn by b (i.e., the upper-bound for color x coincides with In $\texttt{MARK}t(b)(x)$, the number of instances of x withdrawn by b). As a consequence, also the last point of Def. 2 is met. □

Figure 4 shows an example of user-defined transition that may verify Lemma 2, i.e., represent a valid P/T transformation. Conditions $C1$-$C4$ are satisfied, indeed. The condition $C5$ is satisfied if the parameter k on H$\texttt{MARK}t$ is not greater than 3 (if $k < 3$ t is not enabled).


I_O : Arc
O_I : Arc
$2 \cdot \langle p_3, t_2 \rangle$
$\langle p_3, t_2 \rangle$
$3 \cdot \langle p_3, t_1 \rangle + \langle \text{All}, t_2 \rangle$
$\langle p_3, t_1 \rangle$
$2 \cdot \langle p_3, t_1 \rangle$
$\langle p_3 \rangle + 3 \cdot \langle p_1 \rangle$
$2 \cdot \langle p_2, t_1 \rangle + \langle p_3, t_2 \rangle$
H : Arc
$2 \cdot \langle p_1, t_3 \rangle$
t
$\langle p_1 \rangle + 2 \cdot \langle p_2 \rangle$
$[p_1 \neq p_2 \wedge p_2 \neq p_3 \wedge t_1 \neq t_2]$
MARK : P
$\langle p_1, \text{All} \rangle$
$\langle p_2, t_1 \rangle$
IN : Arc
$\langle p_3, t_1 \rangle$
$\text{k} \cdot \langle p_2 \rangle + \langle p_3 \rangle$

class $P = pl\{1..\text{N}\}$ **class** $T = tr\{1..\text{N}\}$ **domain** $Arc = P \times T$

var $t_1 : T$ **var** $t_2 : T$ **var** $t_3 : T$ **var** $p_1 : P$ **var** $p_2 : P$ **var** $p_3 : P$

Fig. 4 A transition possibly verifying Lemma 2

5.4 The net-interface

A crucial aspect of the emulation approach is played by the consistent management of the places `enabList` and `checkList`, which are efficiently updated by exploiting the structural properties of the encoded P/T system. When connecting a transformation rule to the emulator, the semantics of these places must be preserved. In general, this is not at all easy. Therefore, we introduce a *net-interface* that links together the framework's components by transparently handling these aspects.

Formally, places `enabList` and `checkList` must be mutually-exclusive and color-safe to allow the emulation of a P/T system to be carried out consistently. A mostly structural characterization of place color-safeness is given by the following.

Lemma 3 (Characterization of place color-safeness) *Let $t \in T,\ p \in P$.*

If $\boldsymbol{m}[p]$ is type-set and one of the two conditions below is satisfied then: $\boldsymbol{m}[t, b\rangle\boldsymbol{m}' \Rightarrow \boldsymbol{m}'[p]$ is type-set, $\forall b \in \mathcal{D}(t)$.

C1 $H[p,t] \leq All_{\mathcal{D}(t),\mathcal{D}(p)} \wedge W^{+}[p,t] \leq H[p,t]$
C2 $W^{+}[p,t] \leq All_{\mathcal{D}(t),\mathcal{D}(p)} \wedge \exists p' \neq p : \boldsymbol{m}[p] \cap \boldsymbol{m}[p'] = \emptyset \wedge W^{+}[p,t] \leq I[p',t]$

Proof Let $\mathbf{m}[t, b\rangle\mathbf{m}'$. Based on $C1$, $\mathrm{H}[p,t](b)$ is set-type and, because of the enabling rule, $\mathrm{H}[p,t](b) \cap \mathbf{m}[p] = \emptyset$; since $\mathrm{H}[p,t](b)$ includes all the colors which are put in place p by the firing of (t,b), also $\mathbf{m}'[p]$ is type-set. Based on $C2$, $\mathrm{W} + [p,t](b)$, which holds the colors put in p by (t,b), is type-set. Due on the hypothesis on place p', every color of this set is present in p', and not in p. Thus, $\mathbf{m}[p']$ is type-set. □

Color-safeness and mutual-exclusion of places `enabList` and `checkList` can be easily verified on the emulator SN using Lemma 1 and 3. These two invariants are preserved by linking a valid P/T transformation rule t to places `enabList` and `checkList` through the net-interface, which coherently updates their marking based on the annotations of t. That may require a preliminary rewriting of functions In $\mathtt{H}t$, $\mathrm{O}[\mathtt{H},t]$ in disjoint terms that syntactically identify the instances of t that decrease (increase) the multiplicity of inhibitor arcs of an encoded P/T net and those that remove (add) inhibitor arcs. Indeed, decreasing (increasing) the weight of an inhibitor arc provides the opposite effect on the enabling of the linked transition of removing (adding) one such a arc. Such a rewriting is always possible and can be easily automated (we omit the boring technical details). Its semantics is formalized by Def. 7, where subscripts d, r and i, a used for the terms of In $\mathtt{H}t$ and $\mathrm{O}[\mathtt{H},t]$, respectively, denote the different effects described above (i.e., decrease, remove, increase, add).

Definition 7 (Well-linked transformation rule) Let t be a valid P/T transformation. We say that t is well-linked (to the emulator) if and only if:

1. $\mathrm{I}[\mathtt{H},t] = \mathrm{I}[\mathtt{H},t]_r + \mathrm{I}[\mathtt{H},t]_d$ where
   $\mathrm{I}[\mathtt{H},t]_r \cap \mathrm{O}[\mathtt{H},t] \equiv \emptyset \wedge \mathrm{H}[\mathtt{H},t]_{[\overline{\mathrm{I}[\mathtt{H},t]_r}]} - \mathrm{I}[\mathtt{H},t]_r \leq All_{\mathcal{D}(t),\mathcal{D}(\mathtt{H})} \wedge \overline{\mathrm{In}\ \mathtt{H}t_d} \subseteq \overline{\mathrm{O}[\mathtt{H},t]}$
2. $\mathrm{O}[\mathtt{H},t] = \mathrm{O}[\mathtt{H},t]_a + \mathrm{O}[\mathtt{H},t]_i$ where
   $\mathrm{H}[\mathtt{H},t]_{[\overline{\mathrm{O}[\mathtt{H},t]_a}]} \equiv \mathrm{O}[\mathtt{H},t]_a \cap All_{\mathcal{D}(t),\mathcal{D}(\mathtt{H})} \wedge \overline{\mathrm{O}[\mathtt{H},t]_i} \subseteq \overline{\mathrm{In}\ \mathtt{H}t}$

For any *enabled* instance (t,b), the term $\mathrm{I}[\mathtt{H},t]_r$ results in a bag of colors that are all removed (erased) from place H ($\forall x\ \mathrm{H}[\mathtt{H},t](b)(x) - \mathrm{I}[\mathtt{H},t]_r(b)(x) \leq 1$). The term In $\mathtt{H}t_d$, instead, results in a bag of colors that are all left in H, even with lesser multiplicity. The term $\mathrm{O}[\mathtt{H},t]_a$ results in a bag of colors none of which present in place H because the restriction of $\mathrm{H}\mathtt{H}t$ on $\mathrm{O}[\mathtt{H}, t]_a$ is type-set. The term $\mathrm{O}[\mathtt{H}, t]_i$, instead, results in a bag whose elements are all already present in $\mathtt{H}$.

For example, the transition t in Fig. 4 is trivially well-linked. Indeed, any color instance of t adds a *new* weight-2 inhibitor arc to the encoded P/T net, due to the type-set function $\mathrm{H}\mathtt{H}t = \langle p_1, All\rangle$. Therefore, $\mathrm{O}[\mathtt{H}, t] = \mathrm{O}[\mathtt{H}, t]_a$ ($\mathrm{O}[\mathtt{H}, t]_i = \emptyset$). In general, matching Definition 7 may require splitting a transition in a number of mutually-exclusive replicas, as further detailed later on this section. As an example, if in Fig. 4 $\mathrm{H}\mathtt{H}t$ had been *empty* then t should have been split in t' and t'' where $\mathrm{O}[\mathtt{H}, t'] = 2 \cdot \langle p_1, t_3\rangle$, $\mathrm{H}[\mathtt{H}, t'] = \langle p_1, t_3\rangle$, and $\mathrm{O}[\mathtt{H}, t''] = 3 \cdot \langle p_1, t_3\rangle$, $\mathrm{I}[\mathtt{H}, t''] = \langle p_1, t_3\rangle$ (and all the other arc-functions unchanged) to distinguish between the addition of an inhibitor arc and the increase of the weight of an existing one.

Assuming that a valid transformation t matches Def. 7, we can compute the parametric sets of P/T transitions *potentially-enabled* and *potentially-disabled* by any instance of t, whose expressions are described in Table 2 (where t_1 is a projection). The parametric color-set $PD(t)$ represents the P/T transitions for which there are: i) input edges newly added or with an increased weight; or ii) newly added inhibitor edges; or iii) inhibitor edges with a decreased weight. The parametric color-set $PE(t)$ represents, in some sense, the opposite. Note that the sets $PE(t)$ and $PD(t)$ may not be disjoint.

Moreover, the expressions $\overline{\mathrm{W}^{+}[\mathtt{MARK}, t]}$, $\overline{\mathrm{W}^{-}[\mathtt{MARK}, t]}$ represent the parametric sets of P/T places whose marking is increased or decreased, respectively, by an instance of t. By definition, it holds: $\overline{\mathrm{W}^{+}[\mathtt{MARK}, t]} \cap \overline{\mathrm{W}^{-}[\mathtt{MARK}, t]} \equiv \emptyset$.

A P/T transformation t is mechanically linked to the emulator through the net-interface modules shown in Figs. 5 and 6. The link is formally defined by the following output arc-functions, where the function support implicitly denotes the corresponding type-set bag expression (we use the set-notation for readability):

$$\begin{aligned}
\mathrm{O}[\mathtt{potDisab}, t] &= PD(t)\mathrm{O}[\mathtt{potEnab}, t] = PE(t)\\
\mathrm{O}[\mathtt{markInc}, t] &= \overline{\mathrm{W}^{+}[\mathtt{MARK}, t]}\mathrm{O}[\mathtt{markDec}, t] = \overline{\mathrm{W}^{-}[\mathtt{MARK}, t]}\\
\mathrm{O}[\mathtt{beginUpdate}, t] = 1
\end{aligned}$$

Figure 5 shows the module which manages the update of places `enabList` and `checkList` due to a firing of t, by preserving their invariant properties. Figure 6 shows the module performing a preliminary step, which consists of computing all P/T transitions which may be indirectly enabled/disabled due to a P/T transformation occurrence, because of the increase/decrease of the marking of some input/inhibitor place. The two modules are composed through a simple superposition of SN places sharing the name.

Table 2 Parametric sets used to safely update `enabList` and `checkList`

Symbol	Definition	Description
$\mathcal{D}(t) \rightarrow 2^{\mathrm{T}}$		
$PD(t)$	$\overline{t_1 \circ (\mathrm{W}^{+}[\mathtt{IN}, t] + \mathrm{O}[\mathtt{H}, t]_a + (\mathrm{I}[\mathtt{H}, t]_d - \mathrm{O}[\mathtt{H}, t]))}$	T-colors representing P/T transitions potentially disabled by an instance of t.
$PE(t)$	$\overline{t_1 \circ (\mathrm{W}^{-}[\mathtt{IN}, t] + \mathrm{I}[\mathtt{H}, t]_r + (\mathrm{O}[\mathtt{H}, t]_i - \mathrm{I}[\mathtt{H}, t]))}$	T-colors representing P/T transitions potentially enabled by an instance of t.

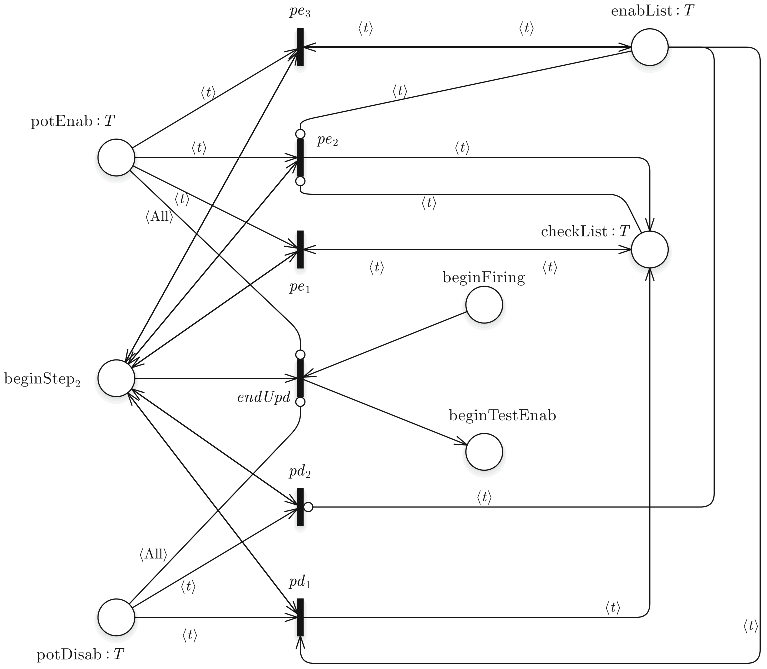


Fig. 5 Part of the net-interface in charge of updating `enabList` and `checkList`

5.5 The evolutionary API

In addition to user-defined transformations, the modeling framework supplies a collection of built-in *read* and *write* primitives, which can be used to sample and safely modify the state/structure of the base-level in procedures implementing the adaptation logics. This evolutionary API allows the internal mechanics of the emulator to be abstracted away from the modeler.

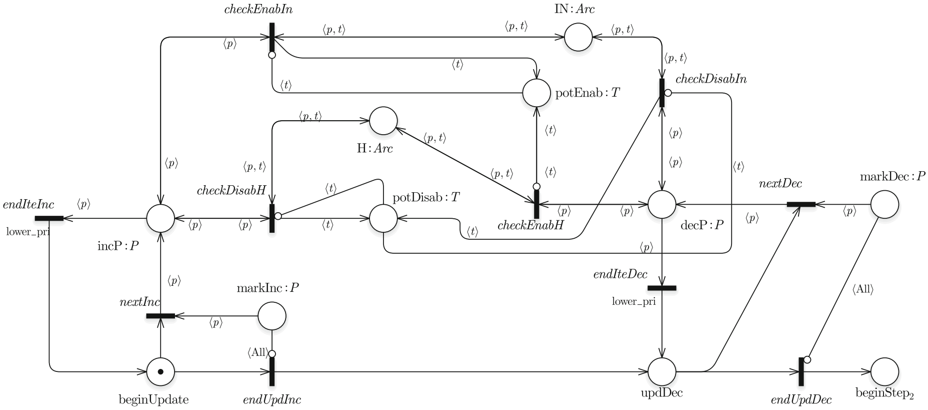


Fig. 6 Part of the net-interface in charge of updating `potEnab`, `potDisab`

Table 3 Read API primitive (sensors)

τ_{in}	τ_{out}	emulator places	rationale
`mark` : P	`mResult`	`MARK` : P	It returns the number of tokens in the given place $p \in P$.
`weightIn` : Arc	`wInResult`	`IN` : Arc	It returns the weight of the given input arc $\langle p, t\rangle \in$ `IN`.
`weightOut` : Arc	`wOutResult`	`OUT` : Arc	It returns the weight of the given output arc $\langle p, t\rangle \in$ `OUT`.
`weightH` : Arc	`wHResult`	`H` : Arc	It returns the weight of the given inhibitor arc $\langle p, t\rangle \in$ `H`.

Definition 8 (API Primitive) An API primitive is a SN subnet N_a composed of *immediate* transitions, such that:

- N_a places include a non-empty subset of **EI** and a disjoint set L of *local* places;
- every N_a transition (linked to **EI**) verifies the conditions of Lemma 2;
- $p_{in} \in L$, with domain P, T, or Arc, such that $\forall t \in N_a$ $\mathrm{W}^+[p_{in}, t] = \emptyset$;
- if $p_{out} \in L$, with $\mathcal{D}(p_{out}) = \mathcal{D}(p_{in})$, then $\exists t \in N_a$ $\mathrm{W}^+[p_{out}, t] \neq \emptyset$;
- if $\mathbf{m}[p_{in}] \neq \emptyset \wedge (p_{out} \in L \Rightarrow \mathbf{m}[p_{out}] = \emptyset)$, we eventually reach $\mathbf{m}'$, $\mathbf{m}'[p_{in}] = \emptyset$;
- $L - \{p_{in}, p_{out}\}$ are covered by P-invariants (i.e., P semi-flows).

The places p_{in}, p_{out} represent a routine's in/out parameters. We can distinguish between *read* and *write* primitives, since only the formers including p_{out}.

The available *read* and *write* API primitives are described on Tables 3 and 4, respectively. For each primitive, the input/output places and the involved **EI** elements are listed.[13] In the following, we use **API** to denote the whole set of the API input/output places.

An example of read primitive is `mark`, which may be used to get the marking of a given P/T place p. It implements a loop which counts for the occurrences of color p in place `MARK`, putting the result (a weighted tuple $\lambda \cdot \langle p\rangle$) in place `mResult`.

An example of write primitive is `addIn`$_k$, depicted in Fig. 7. This primitive adds k occurrences of an input P/T edge, denoted by a color-tuple $\langle p, t\rangle$ in place `addIn`, to the base-level encoding. The operation is atomic: each transition $\mathtt{a}_i$, $i : 0 \ldots k-1$, matches a condition in which there are i occurrences of color $\langle p, t\rangle$ in place `O_I` (which encodes the difference between P/T functions $O - I$), whereas $\mathtt{a}_k$ matches a state in which `O_I` holds at least k occurrences of color $\langle p, t\rangle$. This primitive represents a template, whose actual configuration depends on parameter k. Our case study makes use of `addIn`$_k$ to implement some of the actuators. For instance, the *fault tolerance* procedure uses `addIn`$_1$ to increase the weight of input arc $\langle \mathtt{pl}_i, \mathtt{Assembler}\rangle$ as a part of system's reconfiguration upon the failure of one production line.

It is worth noting that each P/T transformation implemented by the API primitives is valid and it meets Lemma 2. In particular, it preserves mutual exclusion between places `O_I` and `I_O`.

[13] The entire collection of API primitives may be found online, in the public repository containing all the SN models presented in this article.

Table 4 Write API primitive (actuators)

τ_{in}	τ_{out}	emulator places	rationale
$\mathtt{addIn}_k : Arc$	–	$\mathtt{IN} : Arc$, $\mathtt{I\_O} : Arc$, $\mathtt{O\_I} : Arc$	It adds an input arc $\langle p, t\rangle \in Arc$ with multiplicity k.
$\mathtt{addOut}_k : Arc$	–	$\mathtt{OUT} : Arc$, $\mathtt{I\_O} : Arc$, $\mathtt{O\_I} : Arc$	It adds an output arc $\langle p, t\rangle \in Arc$ with multiplicity k.
$\mathtt{addH}_k : Arc$	-	$\mathtt{H}$: Arc	It adds a inhibitor arc $\langle p, t\rangle \in Arc$ with multiplicity k.
$\mathtt{delIn}_k : Arc$	–	$\mathtt{IN} : Arc$, $\mathtt{I\_O} : Arc$, $\mathtt{O\_I} : Arc$	It removes k occurrences of the input arc $\langle p, t\rangle \in Arc$.
$\mathtt{delOut}_k : Arc$	–	$\mathtt{OUT} : Arc$, $\mathtt{I\_O} : Arc$, $\mathtt{O\_I} : Arc$	It removes k occurrences of the output arc $\langle p, t\rangle \in Arc$.
$\mathtt{delH}_k : Arc$	–	$\mathtt{H} : Arc$	It removes k occurrences of the inhibitor arc $\langle p, t\rangle \in Arc$.
$\mathtt{addToken}_k : P$	–	$\mathtt{MARK} : P$	It adds k tokens inside element $p \in P$.
$\mathtt{delToken}_k : P$	–	$\mathtt{MARK} : P$	It removes k tokens from $p \in P$.

6 Validation of the emulating framework

In this section we describe the validation activities conducted upon the emulating framework. In particular we show how *symbolic structural analysis* techniques have been used to validate the behavior of this core component. To make the paper self contained, in Section 6.1 we introduce preliminaries and basic notions. In particular, we introduce the basics of symbolic structural analysis as well as the properties of interest that can be efficiently verified. In Section 6.2, we describe the activity carried out to validate the emulating framework.

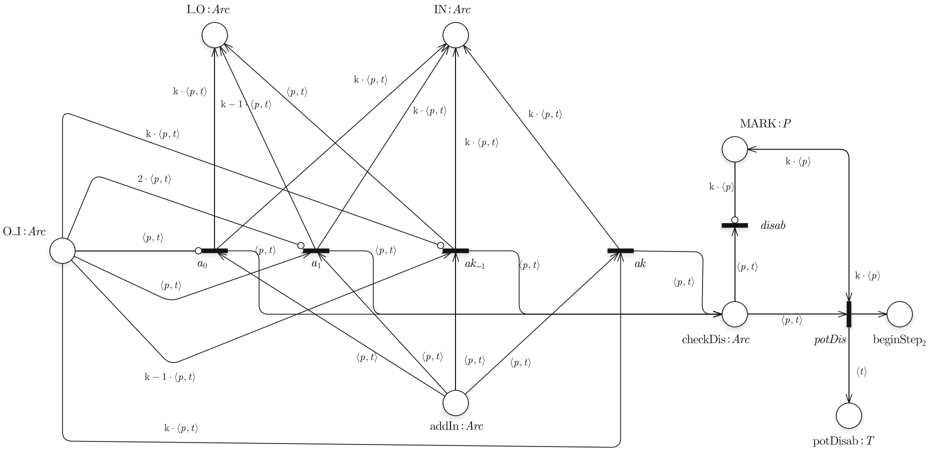


Fig. 7 The $\mathtt{addIn}_k$ API primitive's template

6.1 Basic structural relations

Table 5 reports the symbolic structural relations that we leverage. A symbolic relation between SN nodes e, e' is a map $\mathcal{R}(e, e') : \mathcal{D}(e') \rightarrow 2^{\mathcal{D}(e)}$.

Their formulae involve arc functions and functional operators, in particular, transpose and composition. A description of each relation follows.

Asymmetric Structural Conflict (SC): sets the necessary condition for transition disabling. It maps an instance b' of t' to the set of color instances t that may disable (t', b'), because they withdraw colors (tokens) from an input place of t', or put colors into an inhibitor place of t'. Different instances of the *same* transition may be in self-conflict. In this case, the formula is similar, but for subtracting the identity.

Structural Causal Connection (SCC): defines the necessary condition for enabling. It maps an instance b' of t' to the set of instances of t that may enable (t', b'), because they put colors into an input place of t', or withdraw colors from an inhibitor place of t'.

Structural Mutual Exclusion (SME): (t, b) and (t', b') are in structural mutual exclusion if, whenever one is enabled, the other is not, and vice-versa. This happens if a place p is input for t and inhibitor for t', and the number of a color-token required for the enabling of t is as much, or more, than the limit for that color due to the inhibitor arc function. $SME(t, t')$ maps an instance b' of t' to the set of instances of t that cannot be enabled if (t', b') is enabled. The formula on Table 5 works for type-set functions. P-invariants can be also taken into account. For instance, if $p \ ME \ p'$, then the term $\overline{\mathrm{I}[p,t]}^{Tr} \circ \overline{\mathrm{I}[p,t']} + \overline{\mathrm{I}[p',t]}^{Tr} \circ \overline{\mathrm{In} \ p't'}$ should be added to $SME(t, t')$.

Another interesting possibility offered by SN is the automated verification of *structural invariant*s involving place marking (P semi-flows) or transition sequences (T semi-flows). Even though these invariants do not take account inhibitor arcs and priorities, they allow us to formally and efficiently check properties complementary to structural relations. More in details, focusing on P-invariants, we can distinguish between colored (symbolic) and numerical ones. By using SNEXPRESSION we can check whether a given P-indexed vector of multiset-expressions defines a *colored* P-invariant, and possibly prove the coverage of all place instances. The expressions in the P-indexed vector denote functions from the place color domains to the invariant's domain. Such a vector is a P-invariant (or semi-flow) if, for each transition t, the sum over all places of the compositions of P-invariant's entry $pinv[p]$ with the *linear combination* $\mathrm{O}[p, t] - 1 \cdot \mathrm{I}[p, t]$ results in the *null* function. The invariant marking expression is: $\sum_p pinv[p](\mathbf{m}_0(p))$.

Numerical P-semi-flows (or minimal semi-flow bases) can be checked as well. In fact, SN arc-functions can be rewritten as sums of guarded tuples mapping to constant-size bags, but for those color instances that falsify the guards. As an example, $\langle \mathrm{C}_i - c \rangle$, depending on

Table 5 Symbolic structural relations in SN

Notation	Formula ($Ide_D : D \rightarrow Bag[D]$ is $Ide(d) = 1 \cdot d, \ \forall d \in D$)
$SC(t, t'), t \neq t'$	$\sum_{p \in P} \overline{\mathrm{W}^-[p,t]}^{Tr} \circ \overline{\mathrm{I}[p,t']} + \overline{\mathrm{W}^+[p,t]}^{Tr} \circ \overline{\mathrm{H}[p,t']}$
$SC(t, t)$	$\big(\sum_{p \in P} \overline{\mathrm{W}^-[p,t]}^{Tr} \circ \overline{\mathrm{I}[p,t]} + \overline{\mathrm{W}^+[p,t]}^{Tr} \circ \overline{\mathrm{H}[p,t]}\big) - Ide_{\mathcal{D}(t)}$
$SCC(t, t'), t \neq t'$	$\sum_{p \in P} \overline{\mathrm{W}^+[p,t]}^{Tr} \circ \overline{\mathrm{I}[p,t']} + \overline{\mathrm{W}^-[p,t]}^{Tr} \circ \overline{\mathrm{H}[p,t']}$
$SME(t, t'), t \neq t'$	$\sum_{p \in P} \overline{\mathrm{I}[p,t]}^{Tr} \circ \overline{\mathrm{H}[p,t']} + \overline{\mathrm{H}[p,t]}^{Tr} \circ \overline{\mathrm{I}[p,t']}$
$SME(t, t)$	$(\sum_{p \in P} \overline{\mathrm{I}[p,t]}^{Tr} \circ \overline{\mathrm{H}[p,t]}) - Ide_{\mathcal{D}(t)}$

whether the color bound to c belongs to subclass C_i or not, is either $|\mathrm{C}_i| - 1$ or $|\mathrm{C}_i|$-size. This may be rewritten as $\langle \mathrm{C}_i - c\rangle[c \in \mathrm{C}_i] + \langle \mathrm{C}_i\rangle[c \notin \mathrm{C}_i]$. This is the normal form for bag-expressions used by SNEXPRESSION.

As a consequence, by embedding function-tuple guards in transition guards, any SN transition can be split into a number of equivalent, mutually-exclusive replica,[14] whose I/O arc functions are constant-size, for all valid transition instances. In analogy with low-level PN, a $|P|\cdot|T|$ integer matrix $\mathbf{H}$ is defined, whose $[p, t]$ entry holds $|\mathrm{O}[p, t](b)|-|\mathrm{I}[p, t](b)|$, for any instance b of t. Any *non-null* P-vector $\bar{i}$ which is a positive integer solution of the matrix product $\bar{i}\mathbf{H} = \bar{0}$ is a semi-flow, expressing a conservative law for the tokens flowing through the places corresponding to non-zero entries, abstracting from the token color. A place p is said covered by a semi-flow is the corresponding semi-flow entry is not *null*. A net is said covered if every place is covered. One such net is *bounded*.

6.2 Net-interface validation

The crucial aspect of the emulation framework is played by the consistent management of the places `enabList` and `checkList`. In fact, since transformations are interleaved (not mutually exclusive) with the nominal execution of the base-level layer, they shall be applied safely and avoid any inconsistency due to loss of information and/or addition of spurious information along the interleaved transformation and execution processes. As anticipated in Section 5, such a general property reduces in our context to check whether the places `enabList` and `checkList` are mutually-exclusive and color-safe. In the following we prove that these properties hold by means of the basic structural relations introduced above.

The SN model in Fig. 5 illustrates the part of the net-interface in charge of updating these two places. The model shares the places `enabList`, `checkList`, `beginFiring`, `beginTestEnab` with the emulator. The transitions that put colors into `enabList`, `checkList`, are $\mathtt{pe}_2$, $\mathtt{pd}_1$. The following holds:

$$\mathrm{W}^{+}[\mathtt{checkList}, \mathtt{pe}_2] = \langle t\rangle, \text{ and } \mathrm{W}^{+}[\mathtt{checkList}, \mathtt{pd}_1] = \langle t\rangle \tag{13}$$

Thus, color-safeness of `enabList` is trivially preserved. According to Lemma 1 (A and B), $\mathtt{pe}_2$ preserves mutual-exclusion between `enabList` and `checkList`:

$$\mathtt{enabList}\ SME_{\mathtt{pe}_2}\ \mathtt{checkList} \tag{14}$$

Transition $\mathtt{pe}_2$ preserves color-safeness of `checkList` due to Lemma 3($C1$). Transition $\mathtt{pd}_1$ preserves mutual exclusion between `enabList` and `checkList` and color-safeness of `checkList` since it meets Lemma 1 (A and C) and Lemma 3 ($C2$), respectively. The two basic invariant properties of `enabList`, `checkList` are thus retained.

The *non-empty* structural relations (see Table 5) are listed below by category. In addition to assessing the validity of the emulating framework, we use these properties to reduce the whole model complexity. In fact, from this analysis it emerges that the color domain of transitions is T, but for `endUpd`, which is neutral domain. Therefore, we can ignore conflicts between `endUpd` and the other transitions since $\forall t \neq \mathtt{endUpd}$, $SME(t, \mathtt{endUpd}) = \langle All\rangle$. This means that `endUpd` is mutually exclusive with any instance of every other sub-model transition.

[14] This means that t may be replaced in the SN by $EQ = \{t_i\}$, such that $\mathcal{D}(t) = \mathcal{D}(t_i)$, $\forall t_i \in EQ$, and $\mathbf{m}[t, b\rangle\mathbf{m}' \Leftrightarrow \exists t_i \in EQ : \mathbf{m}[t_i, b\rangle\mathbf{m}'$, $\forall \mathbf{m}, b \in \mathcal{D}(t)$.

Structural Conflict: $\forall i, j, i \neq j\ SC(\mathtt{pe}_i, \mathtt{pe}_j) = \langle t \rangle$; $\forall i, j, i \neq j\ SC(\mathtt{pd}_i, \mathtt{pd}_j) = \langle t \rangle$; $SC(\mathtt{pd}_1, \mathtt{pe}_2) = \langle t \rangle$; $SC(\mathtt{pd}_1, \mathtt{pe}_3) = \langle t \rangle$.

Mutual Exclusion: $SME(\mathtt{pd}_1, \mathtt{pe}_1) = \langle t \rangle$; $SME(\mathtt{pd}_1, \mathtt{pe}_2) = \langle t \rangle$; $SME(\mathtt{pe}_1, \mathtt{pe}_2) = \langle t \rangle$; $SME(\mathtt{pe}_2, \mathtt{pe}_3) = \langle t \rangle$; $SME(\mathtt{pe}_1, \mathtt{pe}_3) = \langle t \rangle$; $SME(\mathtt{pd}_1, \mathtt{pd}_2) = \langle t \rangle$.

Causal Connection: $SCC(\mathtt{pd}_1, \mathtt{pe}_1) = \langle t \rangle$; $SCC(\mathtt{pd}_1, \mathtt{pe}_2) = \langle t \rangle$; $SCC(\mathtt{pe}_2, \mathtt{pe}_1) = \langle t \rangle$; $\forall i\ SCC(\mathtt{pd}_i, \mathtt{endUpd}) = \langle All \rangle$, $\forall j\ SCC(\mathtt{pe}_j, \mathtt{endUpd}) = \langle All \rangle$.

As expected, structural relations between transitions involve only instances of the *same* color. Most of potential conflicts, however, are apparent, because the involved instances are mutually exclusive (possibly due to place mutual exclusion).

Places beginFiring and beginTestEnab are covered by a semi-flow on the emulator model, ensuring that $\mathbf{m}[\mathtt{beginFiring}] + \mathbf{m}[\mathtt{beginTestEnab}] \leq 1$, for each $\mathbf{m}$ reachable from $\mathbf{m}_0$. The sub-model's correct behavior is expressed by the following path property.

Property 4 If $\mathbf{m}[\mathtt{beginStep}_2] = 1 \wedge \mathbf{m}[\mathtt{beginFiring}] = 1$, then we eventually reach $\mathbf{m}'$ such that:

- $\mathbf{m}'[\mathtt{beginStep}_2] = 0 \wedge \mathbf{m}'[\mathtt{beginFiring}] = 0 \wedge \mathbf{m}'[\mathtt{potEnab}] = \emptyset \wedge \mathbf{m}'[\mathtt{potDisab}] = \emptyset$
- $\forall tr \in \mathrm{T}$:

  $tr \in \mathbf{m}'[\mathtt{checkList}]$, if $tr \in \mathbf{m}[\mathtt{potDisab}] \cap \mathbf{m}[\mathtt{enabList}]$;
  $tr \in \mathbf{m}'[\mathtt{checkList}]$, if $tr \in \mathbf{m}[\mathtt{potEnab}] \wedge tr \notin \mathbf{m}[\mathtt{checkList}] + \mathbf{m}[\mathtt{enabList}]$;
  otherwise, $\mathbf{m}[\mathtt{enabList}](tr) = \mathbf{m}'[\mathtt{enabList}](tr) \wedge \mathbf{m}[\mathtt{checkList}](tr) = \mathbf{m}'[\mathtt{checkList}](tr)$.

Proof This property is met if $\mathbf{m}[\mathtt{potEnab}] = \emptyset \wedge \mathbf{m}[\mathtt{potDisab}] = \emptyset$. In this case the only enabled transition (due to SME) is endUpd. Let $\mathbf{m}[\mathtt{potEnab}] \neq \emptyset$: $\forall tr \in \mathbf{m}[\mathtt{potEnab}]$ there is one instance $(\mathtt{pe}_i, tr)$ which is enabled, in mutual exclusion with endUpd, withdrawing a token of color tr from place potEnab (and leaving the neutral token in $\mathtt{beginStep}_2$). Similarly, if $\mathbf{m}[\mathtt{potDisab}] \neq \emptyset\ \forall tr \in \mathbf{m}[\mathtt{potDisab}]$, there is one instance $(\mathtt{pd}_i, tr)$ which is enabled, withdrawing color tr from place potDisab. Thus, the places potDisab and potEnab become eventually empty.

Assuming that $tr \in \mathbf{m}[\mathtt{potDisab}] \cap \mathbf{m}[\mathtt{enabList}]$, the only instances of color tr which are enabled are $(\mathtt{pd}_1, tr)$ and $(\mathtt{pe}_3, tr)$ in case $tr \in \mathbf{m}[\mathtt{potEnab}]$ (according to SME). The latter instance is neither in structural conflict with the former one, nor causally connected to any other instance. Thus, $(\mathtt{pd}_1, tr)$ eventually fires, by moving color tr from enabList to checkList, from where it cannot be removed by any instance.

If $tr \in \mathbf{m}[\mathtt{potEnab}]$, $tr \notin \mathbf{m}[\mathtt{checkList}]$, and $tr \notin \mathbf{m}[\mathtt{enabList}]$, the only transition instances of color tr which are enabled are $(\mathtt{pe}_2, tr)$ and, if $tr \in \mathbf{m}[\mathtt{potDisab}]$, $(\mathtt{pd}_2, tr)$. Also in this case, the latter instance cannot be in conflict (neither directly nor indirectly) with the former one, which eventually fires, by putting color tr into checkList.

In all the other cases, the only transition instances of color tr which may fire preserve the marking of both checkList and enabList, and are not causally connected to any other instance of the same color. □

Considering the model shown in Fig. 6, the following numerical semi-flow holds:

$$Inv = \texttt{beginUpdate} + \texttt{incP} + \texttt{updDec} + \texttt{decP} + \texttt{beginStep2} \tag{15}$$

This equation outlines the control-flow of this portion of the net-interface and it helps us to prove the following path property, which takes into account that transitions `endIteInc`, `endIteDec` (ending the loops managing P/T places whose marking has been possibly increased/decreased, encoded in `marKInc`, `marKDec`) have lower priority.

Property 5 If $\mathbf{m}[\texttt{beginUpdate}] = 1 \wedge \forall p \in Inv - \{\texttt{beginUpdate}\}, \mathbf{m}[p] = 0$ $(\emptyset)$, we eventually reach $\mathbf{m}'$, where $\mathbf{m}'[\texttt{beginStep}_2] = 1 \wedge \mathbf{m}'[\texttt{marKInc}] = \emptyset \wedge \mathbf{m}'[\texttt{marKDec}] = \emptyset$.

The proof in this case is omitted since is comparable to the one of Property 4.

Property 4 and Property 5 together ensure consistent behavior of the whole emulator, once it has been linked to valid P/T transformations through the net-interface. The linking mechanism that allows the whole system to be composed will be described in Section 7.

A last remark concerns two other structural invariant properties of the sub-module in Fig. 6. It preserves color-safeness of places `potDisab`, `potEnab`, that matters for efficiency reasons. In addition, it meets the elementary colored semi-flows $[Ide_{\mathcal{D}(p)}]_p$, $\forall p \in \mathbf{EI}$, meaning that it preserves the emulator's interface.

7 The managing subsystem and module composition

In this section we show how to use the evolutionary API to create *adaptation procedure*s (Section 7.1). We also introduce how to build the target self-adaptive system as a whole by leveraging SN *module composition*. (Section 7.2).

7.1 Managing subsystem

The managing subsystem has a decentralized layout composed of independent *adaptation procedures*. Each procedure implements a feedback control loop, taking into account an adaptation concern, and has a simple characterization. Let $\mathbf{API}_{in}$ be set the set of API input places, $\mathbf{API}_R \subset \mathbf{API}_{in}$ be the input places of read primitives, and, if $p \in \mathbf{API}_R$, $out(p) \in \mathbf{API}_{out}$ be the companion output place.

Definition 9 (Adaptation procedure) An adaptation procedure $\mathcal{P}$ is a SN subnet such that:

- $\mathcal{P}$ places include a non-empty subset of $\mathbf{EI} \cup \mathbf{API}$ and a disjoint set L of *local* places, with $start \in L$
- every $\mathcal{P}$'s transition t matches Lemma 2 (if linked to $\mathbf{EI}$) and:

$$\forall p \in \mathbf{API}_{in}\ \mathrm{I}[p,t] = \emptyset\ \wedge \forall\ p' \in \mathbf{API}_{out}\ \mathrm{O}[p',t] = \emptyset$$
$$(\exists p \in \mathbf{API}_R\ \mathrm{O}[p,t] \neq \emptyset) \Rightarrow \mathrm{H}[out(p),t] = All_{\mathcal{D}(t),\mathcal{D}(out(p))}$$

- L places are covered by semi-flows.

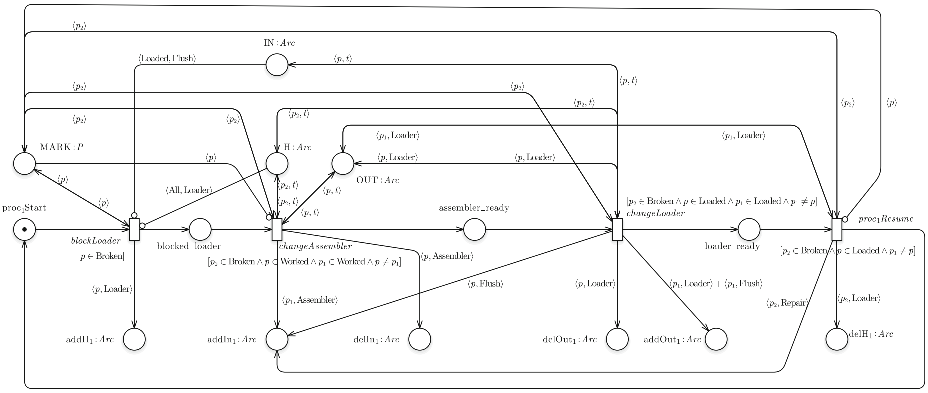
Summarizing, an adaptation procedure may include both visible and immediate transitions, user-defined P/T transformations, as well as calls to the evolutionary API. It can be linked to API input/output places exclusively by means of output/input arc functions, respectively.

Moreover, when invoking read API primitives, the corresponding output places must be empty. The assumption on local places ensure a (local) boundedness, and suggests that they represent a procedure's control-flow.

Figure 8 shows the adaptation procedure (i) *fault tolerance* of the self-healing manufacturing system example. This procedure applies to both the symmetric (SMS) and asymmetric (AMS) fault schema. Its structure meets Definition 9. Being only composed of observable transitions, it is potentially concurrent with the emulated system. The procedure embeds read operations, by directly accessing the emulator's interface **EI** (through pairs of identical I/O arc functions), whereas indirectly carries out changes to the base-level by means of the evolutionary API.

We assume that the manufacturing system (see Section 3) is encoded in the emulator **EI**, and, as shown in Fig. 8, color classes P and T are partitioned into subclasses identifying P/T elements. We do not assume mutual-exclusion between the base-level and the high-level layers. Namely, the manufacturing system continues the production while the adaptation procedures monitor the base-level and enact changes. As discussed in Section 6, the modeling framework ensures that any changes on the base-level leads to a consistent structure. However, the logical correctness of the whole system depends on how the procedures are implemented by the modeler.

As an example, the procedure (i) is triggered when a place $\mathtt{pb}_i$ is marked, meaning that one of the two lines is faulty (e.g., assume line 1). The `blockLoader` transition checks for this by reading the place `MARK` of **EI**. When firing, it also blocks the loader by adding an inhibitor arc between place $\mathtt{pb}_1$ and transition $\mathtt{tlo}_1$, with the $\mathtt{addH}_1$ actuator of the API in order to temporarily isolate the failed production line and rearrange the job on the available line. Then, the procedure adapts the assembler component (i.e., transition `changeAssembler`), by increasing the weight of the edge $\langle \mathtt{pw}_2, \mathtt{ta}_1 \rangle$ through the $\mathtt{addIN}_1$ primitive, and removing the edge $\langle \mathtt{pw}_1, \mathtt{ta}_1 \rangle$ using the $\mathtt{delIN}_1$ primitive. A link between the lines is created, by inserting a new transition, $\mathtt{tfl}_1$, flushing residual row pieces on the faulty line to the other one. Similarly, the procedure adapts the



class $P = \{\mathrm{pin}_1\}$ **is** Input $+\, pl\{1..2\}$ **is** Loaded $+\, pw\{1..2\}$ **is** Worked $+\, pb\{1..2\}$ **is** Broken $+\, \{\mathrm{pa}_1\}$ **is** Assembled $+\, \{\mathrm{pnf}_1\}$ **is** NoFault

class $T = \{\mathrm{tlo}_1\}$ **is** Loader $+\, tl\{1..2\}$ **is** Line $+\, \{\mathrm{ta}_1\}$ **is** Assembler $+\, tfa\{1..2\}$ **is** Fault $+\, \{\mathrm{trs}_1\}$ **is** Restart $+\, \{\mathrm{tfl}_1\}$ **is** Flush $+\, \{\mathrm{tre}_1\}$ **is** Repair **domain** $Arc = P \times T$

var $p : P$ **var** $p_1 : P$ **var** $p_2 : P$ **var** $t : T$

Fig. 8 The SN model of the (i) *fault tolerance* adaptation procedure

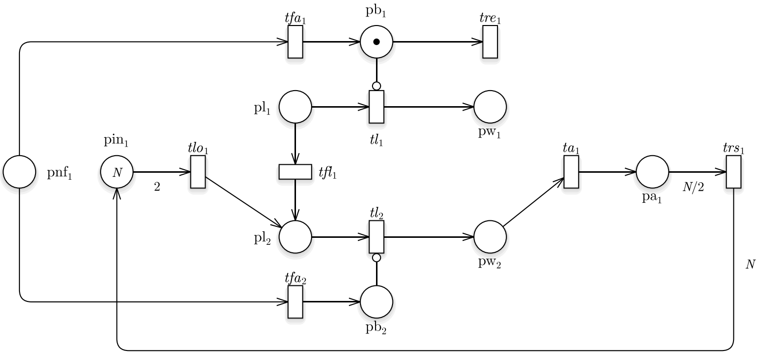


Fig. 9 SMS model after the changes applied by the procedure (i) *fault tolerance*

loader component in order to avoid loading raw pieces into the faulty line. The transition $\mathtt{changeLoader}$ uses the $\mathtt{addOut}_1$ and $\mathtt{delOut}_1$ primitives to increase the weight of edge $\langle \mathtt{pl}_2, \mathtt{tlo}_1\rangle$ and withdraw the edge $\langle \mathtt{pl}_1, \mathtt{tlo}_1\rangle$, respectively. Finally, the managed subsystem is resumed by removing the inhibitor arc between $\mathtt{Broken}$ and $\mathtt{Loader}$. Figure 9 shows the base-level P/T net at the end of the adaptation process. The new transition $\mathtt{tre}_1$, added by procedure (i), simulates the repair activity on the faulty line.

The procedure (ii) *load balancing* brings the manufacturing system back to its nominal layout, after the faulty line has been repaired, and the system has entered a safe state (i.e., the place holding $\mathtt{Worked}$ pieces is empty). The formal specification of this procedure is available with the additional material included on the online repository accompanying this article.

7.2 Module composition

The SN models of our modeling framework (i.e., emulator, net-interface, API primitives, and adaptation procedures) are brought together through *superposition* of shared places (i.e., having same name and equal domain). This represents a convenient and simpler choice compared to transition superposition, since it does not require any rewriting of arc functions.

More precisely, given the emulator net E, a non-empty collection of adaptation procedures $(\mathcal{P}_i)_{i:1\ldots n}$, the API primitives $(N_{ai})_{i:1\ldots k}$ used by any $\mathcal{P}_i$, and the net-interface (Figs. 6, 5), their composition results in a SN model M, whose elements (nodes, color structure, arc functions) are the sum of corresponding elements of involved modules.

To obtain the final SN model we merge the shared places included in the following components:

$$\mathbf{EI} \cup \mathbf{API} \cup \{\mathtt{potEnab}, \mathtt{potDisab}, \mathtt{beginUpdate}\} \tag{16}$$

Subnet composition through superposition of shared places is automated by the ALGEBRA module of GREATSPN introduced by Baarir et al. (2009). The initial marking of M corresponds to the initial marking of its components, defined, as for the fixed parts of the model, in previous sections; all start places of adaptation procedures hold a neutral token.

In order for the different components of the model to correctly synchronize, immediate transitions of different modules are assigned different priority levels,[15] according to a partial order.

[15] The priorities assigned to immediate transitions in each module are *relative*.

Table 6 Structural complexity of models

Model	N	User-defined components (base-level and procedures)	Emulator SN + encoding	Equivalent P/T net (unfolding)
Emulator	-	-	0+29+34+222	0+987+4634+26492
AMS	2	4+37+13+15	28+29+34+222	26+1182+4198+52007
	4	6+37+13+15	30+29+34+222	32+1182+4198+52007
	8	10+37+13+15	34+29+34+222	44+1182+4198+52007
	16	18+37+13+15	40+29+34+222	68+1182+4198+52007
SMS	2	4+38+15+18	32+29+34+222	42+1655+6218+79869
	4	6+38+15+18	34+29+34+222	48+1655+6218+79869
	8	10+38+15+18	38+29+34+222	60+1655+6218+79869
	16	18+38+15+18	46+29+34+222	84+1655+6218+79869

$\pi_{N_1} < \pi_{N_2}$ means that the greatest priority in N_1 is set less than the lowest in N_2

$$\pi_{\mathcal{P}_i} < \pi_{N_{ai}} < \pi_{net\_in} < \pi_{emulator}$$

Table 6 gives empirical evidence of how the emulation-based approach can effectively handle the descriptive complexity of models of self-adapting systems. It reports the number of elements of different Petri Nets that refer to different parts (steps) of a self-adaptive system's model built using the emulation-based approach, by varying the system's parameter (the number N of raw pieces).

The 1st column refers to the user-defined parts of the model, that is, the base-level P/T net used as a case study (both the AMS and the SMS schemas are considered) and the adaptation procedures. The 2nd column refers to the fixed part, i.e., the Emulator SN encoding the base-level. The 3rd column, instead, refers to the P/T system obtained by applying the *unfolding* procedure to the overall SN model resulting from the composition of the fixed and user-defined modules.

The unfolding of an SN model, as described by Chiola et al. (1993), is an equivalent P/T net (with priorities) where each node represents an instance of a corresponding SN node. The unfolding of a stochastic SN is a Generalized Stochastic Petri Net (GSPN) introduced by Chiola et al. (1993).[16]

The numeric values in each column refer to tokens, places, transitions, and arcs, respectively. We observe that the structural complexity of the emulation-based mechanism is more than acceptable (also when considering the module composition) whereas it explodes when using directly P/T nets: the number of net elements needed to model a semantically equivalent system is hundreds of magnitude orders bigger. This huge difference reflects, obviously, the greater expressivity of high-level PN versus classical ones. But it also highlights the enormous difficulty to embed adaptation issues in a plane, non-layered model.

8 Dealing with complexity issues

Major shortcomings of our emulating framework come from the limited data-abstraction capacity of SNs. A high number of immediate transitions are needed to preserve the atomic

[16] The stochastic extension doesn't influence the model's interleaving semantics.

firing semantics of PN transitions, and ensure consistency of the adaptation logic. Legacy solvers/analysers for SNs, integrated in GreatSPN, do not efficiently manage large amounts of immediate transitions in state space generation or discrete-event simulation. In particular, they do not implement any technique to reduce the interleaving of immediate transition color instances, which, when simultaneously enabled, fire in all possible combinations (even when independent from one another). This may result in a combinatorial explosion of vanishing paths/states. GreatSPN state space builders do not even use any on-the-fly reduction (i.e., vanishing paths are erased at the end of the building process). To overcome these issues we discuss in this section on how structural techniques may significantly alleviate the overhead due to immediate transitions. We also consider an orthogonal approach, typical of SN, based on symmetries exploitation.

8.1 Transition interleaving reduction through structural relations

As previously mentioned, the unfolding of a (stochastic) SN results in a GSPN (a P/T net with priorities). GSPN immediate transitions are partitioned into equivalence classes, known as Extended Conflict Sets (ECS) introduced by Balbo (2001).

Each ECS represents a maximal set of transitions of the *same* priority potentially in (direct or indirect) conflict in a marking. The ECS builds on the *indirect structural conflict* described by Balbo (2001), recursively defined using the low-level structural conflict SC, and causal connection SCC (lines 24 and 30 of Alg. 1):

$$t_l \ ISC \ t_k \text{ iff } \pi_l \geq \pi_k \wedge \exists t_j : \pi_j > \pi_k \wedge t_l \ SCC \ t_j \wedge (t_j \ SC \ t_k \vee t_j \ ISC \ t_k) \tag{17}$$

The *symmetric structural conflict* (SSC) is defined as:

$$t_i \ SSC \ t_j \text{ iff } \pi_i = \pi_j \wedge t_i \ SC \ t_j \vee t_j \ SC \ t_i \vee t_i \ ISC \ t_j \vee t_j \ ISC \ t_i \tag{18}$$

Letting $*$ denote the transitive closure, $ECS(t_i) = \{t_j, t_i \ SSC^* \ t_j \wedge \neg t_i \ SME \ t_j\}$. As thoroughly discussed by Balbo (2001), under very general conditions the order in which transitions on different ECS fire does not influence the behavior of a GSPN in terms of its tangible reachability graph (and time semantics). If we chose an arbitrary firing order we may greatly reduce the interleaving of immediate transitions.

We might calculate *symbolic* ECS directly at SN level, by computing the symmetric and transitive closure of symbolic relations (Capra et al. 2015) on Table 5.

As for the emulator, due to some regularity in its structure, we can use a similar, but much simpler technique, based on detection of *apparent conflicts*. If the following holds:

$$SC(t, t') \subseteq SME(t, t') \tag{19}$$

it means that all the color-instances of t which are potentially in conflict with any instances of t' are actually mutually exclusive with them.

For instance, consider the net-interface in Fig. 5. In this model we have checked that the only non-apparent conflict is: $SC(\mathtt{pd}_1, \mathtt{pe}_3) = \langle t \rangle$. That is, any color-instance $\langle tr \rangle$ of $\mathtt{pe}_3$ may be disabled by an instance of the *same* color of of $\mathtt{pd}_1$. According to ECS definition, two such color instances should belong to the same ECS. But, if we take a deeper look, we figure out that they are independent. The situation is described by the diagram in Fig. 10, where: $tr \in \mathbf{m}[\mathtt{potEnab}] \wedge tr \in \mathbf{m}[\mathtt{potDisab}] \wedge tr \in \mathbf{m}[\mathtt{enabList}]$.

You may observe that $(\mathtt{pd}_1, tr)$ disables $(\mathtt{pe}_3, tr)$ and enables $(\mathtt{pe}_1, tr)$. Transitions $\mathtt{pe}_1, \mathtt{pe}_3$, however, are firing-equivalent since:

$$\mathrm{W}^-[p, \mathtt{pe}_1] = \mathrm{W}^-[p, \mathtt{pe}_3] \wedge \mathrm{W}^+[p, \mathtt{pe}_1] = \mathrm{W}^+[p, \mathtt{pe}_3], \forall p \tag{20}$$

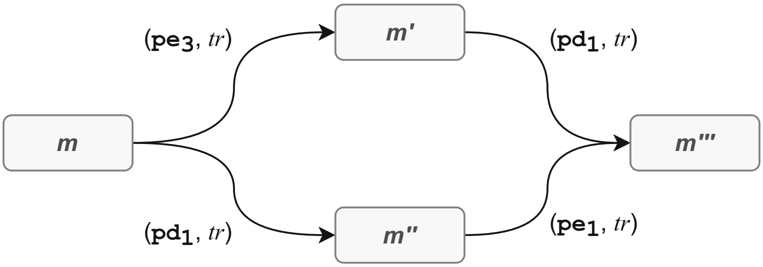


Fig. 10 Conflicting, but actually independent transitions

Moreover, there are no indirect conflicts among net-interface's transition instances, because the higher-priority transitions of the emulator that may be causally connected to or in conflict with them may only be enabled (due to the place-invariant involving place beginFiring) by endUpd, whose firing disables each transition of the net-interface.

We can thus avoid the interleaving among the sub-model's transition instances by assigning each $\mathtt{pd}_i$, $\mathtt{pe}_j$ a different priority.

Using similar arguments, we can check, for instance, that the pairs of transitions checkEnabIn, checkDisabH and checkEnabH, checkDisabIn in Fig. 6 may fire in an arbitrary order, as well transitions a_i of the $\mathtt{addIN}_k$ primitive in Fig. 7.

8.2 Auto-Interleaving Reduction through Ordering/Partitioning Color Classes

The interleaving among color-instances of an immediate transition is another major concern. Experimental evidence shows that a major source of inefficiency is the enabling test (*step i* of Algorithm 1). The bottleneck is represented by the transition toTestT, which non-deterministically selects the next P/T transition to check for enabling. The interleaving of its color-instances can be significantly reduced by defining color class T as *ordered*. This solution, implemented in a second version of the emulator drops the (potential) number of vanishing paths from $\mathcal{O}(|\mathrm{T}|!)$ to $\mathcal{O}(|\mathrm{T}|)$. The same idea can apply to color class P, in the further steps of the emulation. Unfortunately, ordering color classes prevent symmetries from being exploited if, for modeling reasons, they are further partitioned into subclasses, as explained in the next subsection.

The interleaving among color-instances of transition toTestT can be also reduced by observing that $SC(\mathtt{toTestT}, \mathtt{toTestT}) = \emptyset$, i.e., there are no structural auto-conflicts among instances of this transition, and no indirect conflicts caused by higher-priority transition instances. We can thus (automatically) split toTestT into an equivalent set of replicas, each one with a guard ensuring the membership of the color bound to transition's variable t_1 to a given subclass of T. Each replica is assigned a different priority. The emulator potential complexity, in terms of vanishing paths, lowers to $\mathcal{O}(s \cdot max(\{|\mathrm{T}_i|, i : 1 \ldots s\})!)$, where s is the number of subclasses. We can analogously reduce auto-interleaving of other transitions (e.g., testNextIn, testNextH), by exploiting the partition of class P (if any).

8.3 Space complexity reduction through symmetries exploitation

Behavioral symmetries are implicitly expressed in SN models, through the particular syntax of their color annotations. Symmetries are exploited to build a symbolic, quotient reachability graph (SRG) and, in SSN, an associated lumped CTMC, or to drive an efficient discrete-event simulationof SN models.[17]

A *syntactical* state equivalence relation is provided by the *symbolic marking* (SM) notion. A SM is an equivalence class of SN markings: $\mathbf{m}$, $\mathbf{m}'$ are equivalent if and only if one is obtained from the other through a color permutation preserving the static partition and the circular ordering of color classes (if any). As described by Chiola et al. (1993), the SRG built from an initial SM through a symbolic firing rule, retains all the information of the ordinary SN reachability graph.

A SM is defined in terms of *dynamic subclasses*, denoting parametric partitions of each color (sub)class. A dynamic subclass has a cardinality, i.e., it may represent a set of different colors in a (sub)class. Dynamic subclasses of ordered classes are ordered too, representing contiguous sets of colors. A color class which is both ordered and partitioned is like a completely split class, thus the only possible permutation on it is the identity.

An initial SM for the emulator-based model is simply defined from the ordinary initial marking, by replacing color-tokens $\{pl_i\}$ and $\{tr_j\}$ of P, T, with cardinality one dynamic subclasses $\{zp_i\}$ and $\{zt_j\}$, respectively, each referring to a given subclass in the case of a static partition of P, T. The nodes (SM) of the resulting SRG, may thus represent classes of *isomorphic* P/T systems.[18] Checking graph isomorphism is in general complex.[19] In SN encoding of bipartite graphs, it corresponds to bringing a SM into a *canonical form*, as described by Chiola et al. (1993). Most of the inefficiency of legacy SRG algorithm is due to a brute-force enumeration of color permutations done during SM canonization. Therefore, although the SRG may result in a more compact structure than the ordinary RG, its execution time may be incomparably higher.

8.4 Experimental results on complexity

Table 7 reports some experimental data. All results reported henceforth have been obtained running GreatSPN on Fedora Gnu/Linux release 25 operating system, using a machine equipped with a 2.5 GHz Intel Core i5 processor, and 16 GB 1600 MHz DDR3 RAM.

Data refers to the state space building process of the emulating framework encoding the SMS example (without adaptation procedures) by varying the model size N (i.e., number of raw pieces). In case the computation succeeds, the Table reports the execution time and the corresponding number of Tangible Marking (TM) and Vanishing Markings (VM). Otherwise it reports either Fail (i.e., out of memory) or TimeOut (i.e., 3 hours). Data has been collected by using different versions of the emulating framework (Emu$_1$-Emu$_4$) that implement different techniques to reduce the complexity discussed above.

The overhead due to immediate transitions clearly emerges. Nevertheless, it is possible to quantify the contribution of each reduction technique used to optimize the emulating framework. It is worth noting that, without any optimization, we have been able to solve models up to $N = 8$. In this case, we observe a number of VMs 9 times larger than what we obtain by using the Emu$_1$ version.

We also observe that the reduction of VMs achieved by ordering class T is comparable with the one obtained by partitioning T, whereas the latter adds up to the reduction due to

[17] In this context we only consider analytical solutions.

[18] An isomorphism between P/T systems is a permutation of the corresponding bipartite graph's nodes preserving connections and marking.

[19] Graph isomorphism belongs to NP, but is thought to be neither P nor NP-complete.

Table 7 SMS state space building using different complexity reduction techniques

	EMU$_1^a$		EMU$_2^b$		EMU$_3^c$		EMU$_4^d$	
N	time (s)	TM/VM	time (s)	TM/VM	time (s)	TM/VM	time (s)	TM/VM
4	<1	60/8514	<1	60/4438	<1	60/5227	2	35/2486
8	<1	315/55047	<1	315/30636	<1	315/34001	17	175/16533
16	21	2475/506073	12	2475/297978	13	2475/316108	$\approx$ 4000	1290/149826
32	295	26163/6008669	179	26163/3652670	191	26163/3792654	TIMEOUT	-
40	863	58443/13868559	468	58443/8476468	473	58443/8783701	TIMEOUT	-
48	FAIL	–	838	82800/12195696	838	82800/12635744	TIMEOUT	–

[a] Transition Interleaving Reduction

[b] Transition Interleaving Reduction + Ordered Classes

[c] Transition Interleaving Reduction + Partitioned Classes

[d] Transition Interleaving Reduction + Partitioned Classes + Symmetries Exploitation

the symmetries. Experiments confirm that the SRG structure is smaller compared to the RG. Nevertheless, the time complexity of the SRG algorithm is higher as discussed above.

When considering the SMS example in Fig. 2 (two parallel production lines), the exploitation of symmetries reduces the state space by a factor of ~ 2 (according to the model's layout). Nonetheless, symmetries become crucial when considering systems having a high number of replicated components. For instance, consider the parametric model represented by the SN in Fig. 11 composed of M modules (i.e., production lines), working in parallel and synchronizing at a given point ($M = 2$ corresponds to the SMS example in Fig. 2).

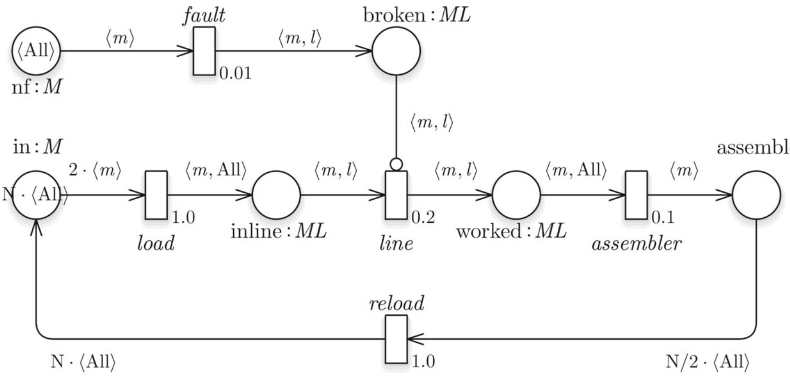
When M grows up, the exploitation of symmetries becomes the only effective way to deal with the model's scalability. The intrinsic combinatorial complexity of one such model is untreatable otherwise (to give an idea, for $M = 8$ there are several dozens millions of ordinary states against just a few thousands of symbolic ones). Once again, the drawback here is the inefficiency (in terms of execution time) of the legacy SRG builder of GreatSPN, whose re-engineering is planned as a part of our future work.

9 Formal verification

In this section, we discuss significant verification activities that can be carried out on adaptive systems modeled with our framework. Since this approach relies on SNs, we can leverage a wide range of consolidated techniques. The techniques discussed in this section have been applied to our selected case study (both ASM and SMS schemas).

9.1 Timing analysis

As anticipated in Section 4, the emulator model can be used to model/simulate also stochastic PNs (SPNs), Each SPN transition t_k is associated with a *rate* $\rho_k \in \mathbb{R}^+$ representing the parameter of a non-negative exponential density function from which the firing delay of t_k is sampled, once it is enabled in a marking. Rates can be marking-dependent. This



class $M = m\{1..\max\}$ **class** $L = \{l_1, l_2\}$ **var** $m : M$ **var** $l : L$

domain $ML = M \times L$ $\mathrm{N} = 4$ $\max = 2$

Fig. 11 SN model of a SMS composed of M symmetric modules

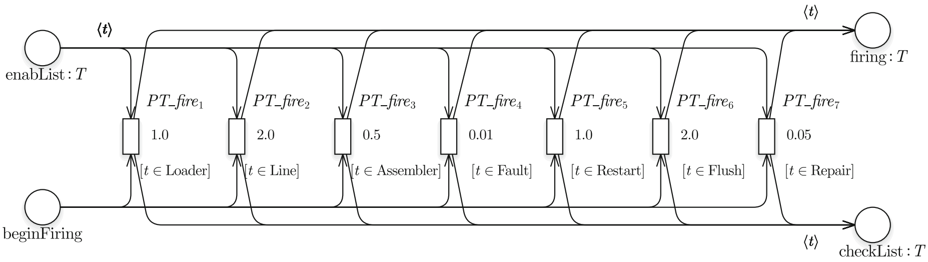


Fig. 12 `PT_fire` partitioning to encode the SPN manufacturing system

enhances the expressiveness of SPNs. As a result, a SPN is isomorphic to a Continuous Time Markov Chain (CTMC) – see Gagniuc (2017) – whose states are the SPN's reachable markings. The $[i, j]$ component of the CTMC generator matrix ($i \neq j$) is the sum of rates of transitions that lead from m_i to m_j. Performance indices such as transition throughputs and token distribution in places can be computed from either the transient or (if the model is ergodic) the steady-state probability vector. The CTMC is automatically derived from the SRG structure. This latter model can be efficiently solved (instead of the original one), to get performance indices.

To emulate a SPNs, we need to consider the emulator as a stochastic SN (SSN), formerly known as SWN, introduced by Chiola et al. (1993). The timed semantics of a SSN is that of its unfolding, i.e., a Generalized SPN. Observable transitions (called *timed*) are associated with exponential rates, as in SPN. Immediate transitions fire in zero-time with priority over timed ones, and are assigned *weights* used to solve in a probabilistic way possible conflicts arising in vanishing paths. As a result, the semantics of a SSN is a CTMC isomorphic to its TRG. The probabilities associated with vanishing paths determine, together with timed transition rates, the entries of the CTMC generator matrix. If an initial symbolic marking is set for a SSN, then it is possible to derive a *lumped* CTMC isomorphic to the tangible part of its SRG.

Rates/weights can be associated with SSN transition *color-instances*, or be marking-dependent, according to the static partitioning of color classes.[20] In the emulator module, arbitrary rates can be attached to immediate transitions color-instances of the model's fixed parts without affecting its stochastic behavior.

To encode a SPN in the emulator, we may have to partition the emulator's color class T, so that we can distinguish groups of P/T transitions with different firing rates. P/T transitions are in fact encoded as color-instances of the SN *timed* transition `PT_fire`.

Figure 12 shows how the `PT_fire` transition of the emulator has been automatically split in order to encode a stochastic version of the MS running example. Each individual `PT_fire`$_i$ instance has a different firing rate, and corresponds to the firing of a subclass of T.

Table 8 shows data obtained by analyzing the self-healing manufacturing systems (both *AMS* and *SMS*), including the adaptation procedures (managing line-faults and subsequent repairs), with different model size N. The reachability graph built by using the optimized EMU3 version (i.e., transition interleaving reduction and partitioned classes) holds one maximal strongly-connected component (i.e., the initial marking is a home state), thus there is a steady-state solution of the corresponding CTMC.

[20] The current GREATSPN release doesn't support these features

Table 8 Reachability graph building and transition throughputs

Model	N	$\|RG\|$ (TM/VM)	time (s.)	$\|SRG\|$ (TM/VM)	Time (s.)	Throughput	Time (s.)
AMS							
	2	55/3484	0.15	55/3484	3.40	0.20074	0.66
	4	184/13906	2.31	184/13906	10.06	0.26590	1.12
	8	985/91586	8.04	985/91586	64.11	0.33175	5.36
	16	7964/886842	77.11	7964/886842	622.91	0.38866	49.71
SMS							
	2	92/6708	1.71	48/3437	5.36	0.19532	0.93
	4	276/23268	2.79	184/13906	10.06	0.25852	1.88
	8	1289/131761	10.12	662/66663	91.66	0.32164	5.59
	16	9103/1114027	99.05	4634/561461	816.71	0.37521	62.27

The first significant remark here is that the RG and the SRG in the AMS example have the same size. In this example, we do not reduce the space complexity with the SRG structure. This result reflects the structural properties of this model (i.e., it does not have behavioral symmetries). The SMS is composed of two symmetric parts (i.e., two faulty production lines). Here, the size of the SRG is reduced by a factor of $\sim$ 1.9 (considering both the tangible and the vanishing markings). The time instead increases by a factor of $\sim$9,

varying from a few milliseconds (0.15 s. with $N = 3$) to a few minutes (208.11 s. with $N = 6$). The *throughput* column in Table 8 shows the average throughput of the *Assembler* component (i.e., the number of firings of the transition $\mathtt{ta}_1$ in a time unit) of the encoded SPN. This value is congruently the same for the ordinary CTMC and the lumped one derived from the SRG. The last column of Table 8 reports the time required to compute the throughput value for all the transitions of the model. This value ranges from a few milliseconds (0.66 s. with $N = 2$) to a few seconds (49.71 s. with $N = 16$).

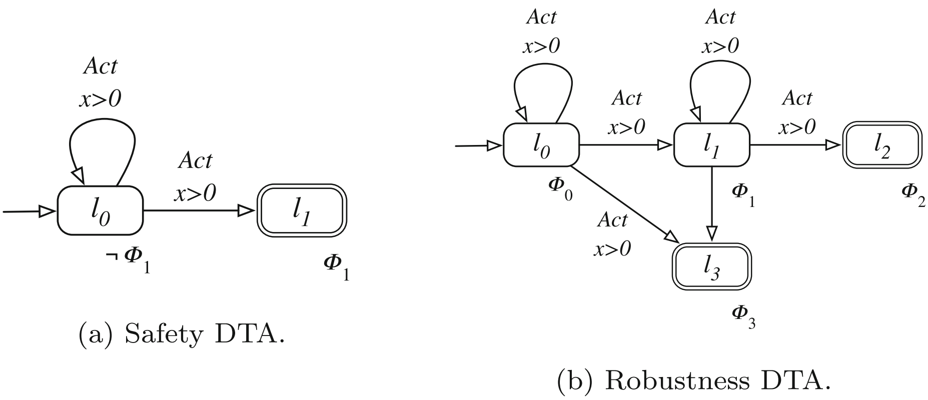
9.2 Model checking

The $MC4CSL^{TA}$ model checker, introduced by Amparore and Donatelli (2010), has been used to verify the correctness of our case studies with respect to design-time requirements. $MC4CSL^{TA}$ is a probabilistic model checker embedded in GreatSPN for the stochastic logic CSL^{TA}. In the following, we provide a brief overview on the CSL^{TA} stochastic logic to make the article self contained. We refer the reader to Amparore and Donatelli (2010) for a more details.

The CSL^{TA} stochastic logic for CTMCs is characterized by the possibility of specifying path formulas through a single-clock Deterministic Timed Automata (DTA). Given $\lambda \in [0, 1]$ a probability value, $p \in AP$ an atomic proposition, and $\bowtie \in \{\leq, <, >, \geq\}$ a comparison operator, a CSL^{TA} state formula Φ is defined as follows:

$$\Phi ::= p \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathcal{S}_{\bowtie\lambda}(\Phi) \mid \mathcal{P}_{\bowtie\lambda}(\mathcal{A}),$$

where $\mathcal{A}$ is a single-clock DTA. CSL^{TA} uses DTAs to specify (timed) accepted paths and the model checker goal is to compute the probability of the set of accepted paths. A DTA $\mathcal{A}$ is composed of a set of states and a set of edges (e.g., see Fig. 13a). Each DTA is equipped with a clock (usually named x), that runs constantly and whose value increases linearly over time. Edges describe the transition relation and can be labeled with a clock and action



(a) Safety DTA.

(b) Robustness DTA.

Fig. 13 DTAs used in the CSL^{TA} properties (21) and (22), respectively

constraints. The DTA in Fig. 13a has two states l_0 and l_1. The state l_0 is initial, and l_1 is final. An edge with a constraint in the form $x = \alpha$ is a *boundary edge* (marked with the $\sharp$ symbol), and is triggered by elapsed time. An edge with a constraint $\alpha < x < \beta$ is an *inner edge* and is triggered by a transition firing in the PN model. States can have an associated boolean formula which predicates on the marking of the PN model.

Formal requirements in our framework can be expressed as general *reachability*, *safety*, *liveness* properties and *invariant*s using the CSL^{TA} stochastic logic. The properties can refer either to the behavior of the managing (i.e., the adaptation procedures) or the managed (i.e., the encoded base-level model) subsystem.

Safety Safety properties can predicate over either the nominal behavior or the adaptation process. For instance, we verified that the manufacturing systems (considering all their possible evolutions) does not lose raw pieces during the production. Equation 21 shows the formula used to verify the property. We use #p to denote the number of tokens inside the place p.

$$\mathcal{P}_{\leq 0}\ \textit{safety}\ (\Phi_0 := \#\texttt{pin}_1 + \#\texttt{pl}_1 + \#\texttt{pl}_2 + \#\texttt{pw}_1 + \#\texttt{pw}_2 + 2 \cdot \#\texttt{pa}_1 \neq N), \qquad (21)$$

Figure 13a shows the *safety* DTA embedded in this formula. The *Act* label represents a *wildcard action* and it stands for “any firing transition”. Since in the current example we do not take into account time, we leave a default check $x > 0$ on the clock variable x. This means that the actions are executed at any time greater than zero (i.e., *Zeno* behavior is not admitted).

Robustness An example of robustness property verified on the manufacturing systems follows. If no failures occur (i.e., $\#\texttt{pb}_1 = 0$, $\#\texttt{pb}_2 = 0$), the production lines work correctly and they eventually assemble all the raw pieces (i.e., $\#\texttt{pin}_1 = 0 \wedge \#\texttt{pa}_1 = N/2$). On the contrary, if a failure occurs, the system is able to reconfigure itself and restore its own nominal behavior. The CSL^{TA} formula follows. The embedded *robustness* DTA is reported in Fig. 13b.

$$\begin{aligned}\mathcal{P}_{\geq 1}\ \textit{robustness}\ (\Phi_0 &:= \#\texttt{pin}_1 \geq 0 \wedge \#\texttt{pb}_1 = 0 \wedge \#\texttt{pb}_2 = 0,\\ \Phi_1 &:= \#\texttt{pb}_1 > 0 \vee \#\texttt{pb}_2 > 0, \Phi_2 := \#\texttt{pb}_1 = 0 \wedge \#\texttt{pb}_2 = 0,\\ \Phi_3 &:= \#\texttt{pin}_1 = 0 \wedge \#\texttt{pa}_1 = N/2) \qquad (22)\end{aligned}$$

Liveness An example of liveness property verified on the manufacturing systems is reported in (23). It states that if a failure occurs, the *fault tolerance* procedure always statrs (i.e., $\#\texttt{procedure}_1 Start > 0$). Figure 14a shows the embedded *liveness* DTA.

$$\mathcal{P}_{\geq 1}\ \textit{liveness}\ (\Phi_0 := \#\texttt{pb}_1 > 0 \vee \#\texttt{pb}_2 > 0,\ \Phi_1 := \#\texttt{procedure}_1 Start > 0) \tag{23}$$

A similar liveness property can be verified by targeting the *load balancing* adaptation concern. Namely, we verified that the *load balance* procedure always starts when a broken production line has been fixed.

Furthermore, we verified the *adaptation integrity constraint* (i.e., a common meta-property often verified in self-adaptive systems (Zhang and Cheng 2006)). This meta-property states that once the adaptation starts, it must eventually complete. Thus, the adaptation process eventually reaches its own final state and the desired changes are applied to the base-level. In the manufacturing systems, we verified this constraint by using the liveness formulas (24) and (25). The two CSL^{TA} properties target the two adaptation procedures, respectively.

$$\mathcal{P}_{\geq 1}\ \textit{liveness}\ (\Phi_0 := \#\texttt{procedure}_1 Start > 0,\ \Phi_1 := \#\texttt{resumeProc}_1 > 0) \tag{24}$$

$$\mathcal{P}_{\geq 1}\ \textit{liveness}\ (\Phi_0 := \#\texttt{procedure}_2 Start > 0,\ \Phi_1 := \#\texttt{resumeProc}_2 > 0) \tag{25}$$

Timed properties Up to this point, we introduced a number of absolute properties, i.e., verified either with probability ≥ 1 (i.e., always true) or with probability ≤ 0 (i.e., always false). In fact, we considered so far a base-level encoded as P/T net. Nevertheless, a base-level encoded as stochastic PN (see Section 9.1) opens up the possibility of verifying probabilistic (timed) properties. Equation 26 reports an example of timed property verified upon the stochastic *AMS* systems.

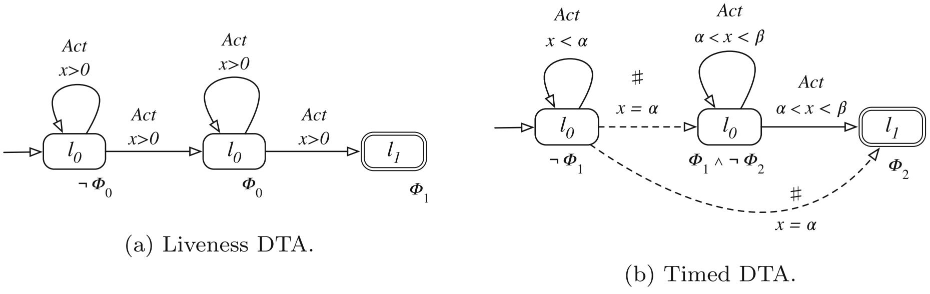
$$\mathcal{P}_{>0.9}\ \textit{timed}\ (\alpha := 1, \beta := 10, \Phi_0 := \#\texttt{pb}_1 > 0 \vee \#\texttt{pb}_2 > 0,\\ \Phi_1 := \#\texttt{pb}_1 = 0 \wedge \#\texttt{pb}_2 = 0) \tag{26}$$

The probability states that the probability fixing a failed line, within 10 time units, is greater than 0.9. The embedded *timed* DTA is shown in Fig. 14b.

Another example follows.

$$\mathcal{P}_{>L}\ \textit{timed}_2\ (\alpha := X, \Phi_0 := \#\texttt{pin}_1 = N, \Phi_1 := \#\texttt{pa}_1 = N/2) \tag{27}$$

This latter property has been used to verify that the time required by the system to assemble all the raw products, in X time unit, is greater than L. Figure 15 reports the embedded *timed*$_2$ DTA.



(a) Liveness DTA.

(b) Timed DTA.

Fig. 14 DTAs used in the CSL^{TA} properties (23), (24), (25), and (26)

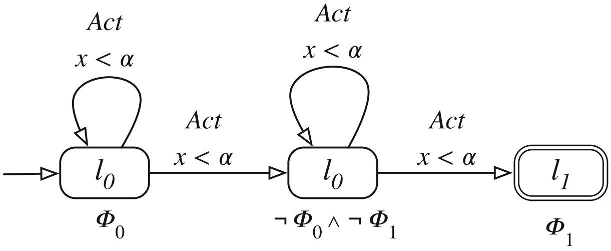


Fig. 15 DTA used in the CSL^{TA} property (27)

Table 9 shows, for each property, the total execution time and space. Time values refer to resources required by both on-the-fly state space construction and computation of the forward steady-state solution of the

SN model. The last time column shows the average time value required by these two (separated) activities. Results reveal that the amount of time required to compute the solution in both the two cases (*AMS* and *SMS*) is negligible and very low if compared with the build time. The build time varies from few milliseconds ($\sim$ 0.16s./$\sim$ 0.46s. with $N = 2$) to few minutes ($\sim$ 44s./$\sim$ 90s. with $N = 16$). The last column shows the average memory usage during the verification activities. It is worth noting that the total amount of memory is small even though the state space increases (especially in the number of vanishing markings) when increasing the problem size N. Average values remain below 1MB ($\sim$ 1KB/$\sim$ 2KB with $N = 2$ to $\sim$ 120KB/$\sim$ 160KB with $N = 16$).

10 Related work

The approach described in this paper is part of our ongoing research activity on formal specification and verification of decentralized self-adaptive systems, as described by Camilli et al. (2018), Capra and Camilli (2018), Camilli et al. (2019), and Capra and Camilli (2020). Self-adaptation has been widely studied from different perspectives and different

Table 9 Verification time and space

		Time (s.)								Space (KB)
Model	N	(5)	(6)	(7)	(8)	(9)	(10)	(11)	avg build+sol	avg memory
AMS										
	2	0.16	0.16	0.17	0.16	0.16	0.15	0.16	0.16+0.00	1
	4	0.64	0.59	0.62	0.62	0.63	0.58	0.58	0.61+0.00	3
	8	4.06	4.05	4.05	4.04	4.03	4.05	4.27	4.06+0.01	16
	16	44.9	45.10	43.77	43.80	44.62	43.24	44.28	43.88+0.37	121
SMS										
	2	0.43	0.49	0.43	0.47	0.48	0.47	0.46	0.46+0.00	2
	4	1.52	1.53	1.54	1.54	1.52	1.50	1.55	1.52+0.01	6
	8	9.07	9.05	9.71	9.10	9.13	9.14	9.10	9.14+0.04	25
	16	86.40	98.96	92.61	91.31	93.13	88.81	84.31	90.49+0.29	160

research communities, such as software architectures, programming languages, software-engineering, and formal methods, to name a few. We let the reader refer to Salehie and Tahvildari (2009), Weyns et al. (2012), de Lemos et al. (2017), and Arcaini et al. (2017) for a more comprehensive overview on the landscape of self-adaptive systems. Here, we limit the comparison to formal methods for specifying and verifying self-adaptation, with particular focus on models of concurrency and DESs, such as PN-based approaches. In fact, although PNs represent a sound and expressive model of concurrency and distribution, they cannot represent in a natural way structural changes able to express self-perceiving and self-adaptation procedures (Reisig 1985). Several attempts to face this critical issue gave rise to new PN-based formalisms, in which an enhanced modeling power is not always accompanied by adequate analysis techniques.

The formal framework introduced in this article has been mainly influenced by different existing approaches able to describe DESs with evolvable and/or adaptable behavior. A different, even if somehow related, paradigm based on pure *SPEC*-inscribed (algebraic) PNs – see Reisig (1991) – has been introduced by Capra (2016). The framework builds on a compact algebraic net which emulates the behavior of any P/T system encoded as a marking of the net. The model is completed by a library of atomic transformations (similar to graph-rewriting rules) implemented as (parametric) transitions, that one can use and compose with the emulator net to describe self-adaptation procedures. The OBJ specification language, introduced by Goguen et al. (1988) is adopted to inscribe the high-level nets. Though inspired by the same principles and sharing most of the goals, the approach introduced in this work is uniform and relies on well established formal methods. This choice permits consolidated analysis techniques to be leveraged. In fact, the main drawback of the algebraic emulator (and many other state of the art approaches) is the lack of supporting software tools. Currently available tools for algebraic PNs are based on too complex higher-order languages to non-expert users. Moreover, they often have a non standard and less intuitive semantics than the original one of pure algebraic specifications, introduced by Reisig (1991).

The use of “higher-order” tokens in PNs dates back to the approach presented by Valk (1998), where both system and object nets are Condition/Event systems. The synchronization between these entities is performed though special interface transitions. While preserving most principles of elementary nets theory, this technique does not allows dynamical transformations to be represented. A survey of approaches that combine “higher-order” tokens and the features of object-oriented programming has been presented by Valk (2004). In these models objects are instances of classes and object transformations are specified by methods.

However, most of these formalisms do not have a clear denotational semantics, thus limiting the applicability of classical analysis techniques. Reference Nets, introduced by (Cabac et al. 2005), are the representative of this class of formalisms, being supported by a Java software tool called RENEW. Another modeling approach, based on object-oriented Petri nets, has been introduced by Meng (2010) to describe reconfigurable manufacturing systems. This model integrates object-oriented methods, stepwise refinement ideas and Petri nets together. In particular the authors introduce the concepts of macro-place, used to model the aggregation of many processes, and macro-transition, used to link all the related macro-places. This work focuses on the modeling activity. Validation and verification of such a models are currently not supported. Algebraic Higher-Order (AHO) nets, described by Hoffmann and Mossakowski (2002), are an extension of algebraic nets where the employed algebraic specification language HASCASL is higher-order. Tokens in AHO-nets represent P/T nets, that move across places, and that can be transformed using a restricted set of algebraic net-operators applied by firing transitions. AHO-nets have been proposed as a unifying framework for the "nets within nets" paradigm. An interesting evolution of this paradigm, based on AHO-nets and on the main ideas of graph transformation systems, has been introduced by Hoffmann et al. (2005a, b), where the token game is integrated with rule-based transformations of P/T nets. An algebraic extension of object nets called Higher Order Recursive Nets is proposed in Köhler-Bußmeier (2009), where an algebraic structure which refers to the topology of net-tokens is introduced, providing operators for nets composition. Net Rewriting Systems (NRSs) have been introduced by Llorens and Oliver (2004). This formalism represents a model of the dynamic changes in the structure of Petri nets. However such a formalism do not define analysis techniques able to formally verify design-time requirements. Reconfigurable Petri Nets (RecPNs) represent a subclass of NRS where reconfigurability is restricted to the flow relation (i.e., changes on place/transition elements are not feasible). In RecPNs, a system configuration is described as a P/T net and a change in the configuration is described as a graph rewriting rule. Verification algorithms here, are based on a translation process between a RecPNs model to an equivalent P/T net, so that the classical verification methods of P/T nets are applicable. An extension of NRSs (i.e., Improved RNSs) has been introduced by Li et al. (2009) to overcome property decidability issues. In this extension, three basic properties (liveness, boundedness, and reversibility) can be verified.

Other related approaches grounded on reflective frameworks have been introduced by Capra and Cazzola (2006) and Camilli et al. (2018). In particular, Reflective Petri nets (RPNs) aim at specifying and simulating evolving P/T systems through a meta-modeling layer composed of *guard*s and *strategie*s, provided in term of high-level Petri nets. The meta-modeling layer provides the ability to describe conditions (i.e., guards) that activate specific routines (i.e., strategies) able to change the structure of the lower layer (i.e., the P/T net). The main drawback of RPNs, once again, is the lack of supporting analysis techniques and software frameworks/tools.

The development of (single layer) PN-based approaches to specify and verify self-adaptive systems has been addressed by Zhang and Cheng (2006). An extension of this latter work, which takes into account also temporal concerns in real-time self-adaptive systems has been introduced by Camilli et al. (2015, 2018). The key idea of these modeling approaches is to unfold the nominal and the evolutionary behaviors of a self-adaptive systems in a single modeling layer, realizing a separation of concerns by means of *zone*s (i.e., PN subnets). However these approaches allow to represent the adaptation from an abstract perspective, meaning that they do not allow the specification of adaptation procedures and the conditions under which the system is required to adapt itself. Although they are well supported by both design-time – see Bellettini et al. (2012) and Camilli (2014) – and runtime – see Camilli et al. (2017a) – analysis techniques, these approaches have limited modeling capability and they do not allow for an easy integration of structural changes in the model itself.

Other models of concurrency have been tailored to represent adaptable and/or evolvable systems. Process Algebras, such as Communicating Sequential Processes (CSP), have been extended in Bartels and Kleine (2011) to model systems able to react upon external stimuli by changing their internal behavior (e.g., to recover from errors). In this sense, self-healing systems can be viewed as a reactive systems that adapts itself in response to external inputs. Reconfiguration of systems is also supported by the work presented by Allen et al. (1998). Systems are described using the Architecture Description Language (ADL) *Dynamic Wright*. The formal semantics of the ADL is defined by a translation to CSP. Hence, formal properties on the architecture of the system can be verified on the level of CSP. These works focus on architectural aspects of adaptable system design and does not consider functional aspects of the system behavior.

11 Conclusion and future directions

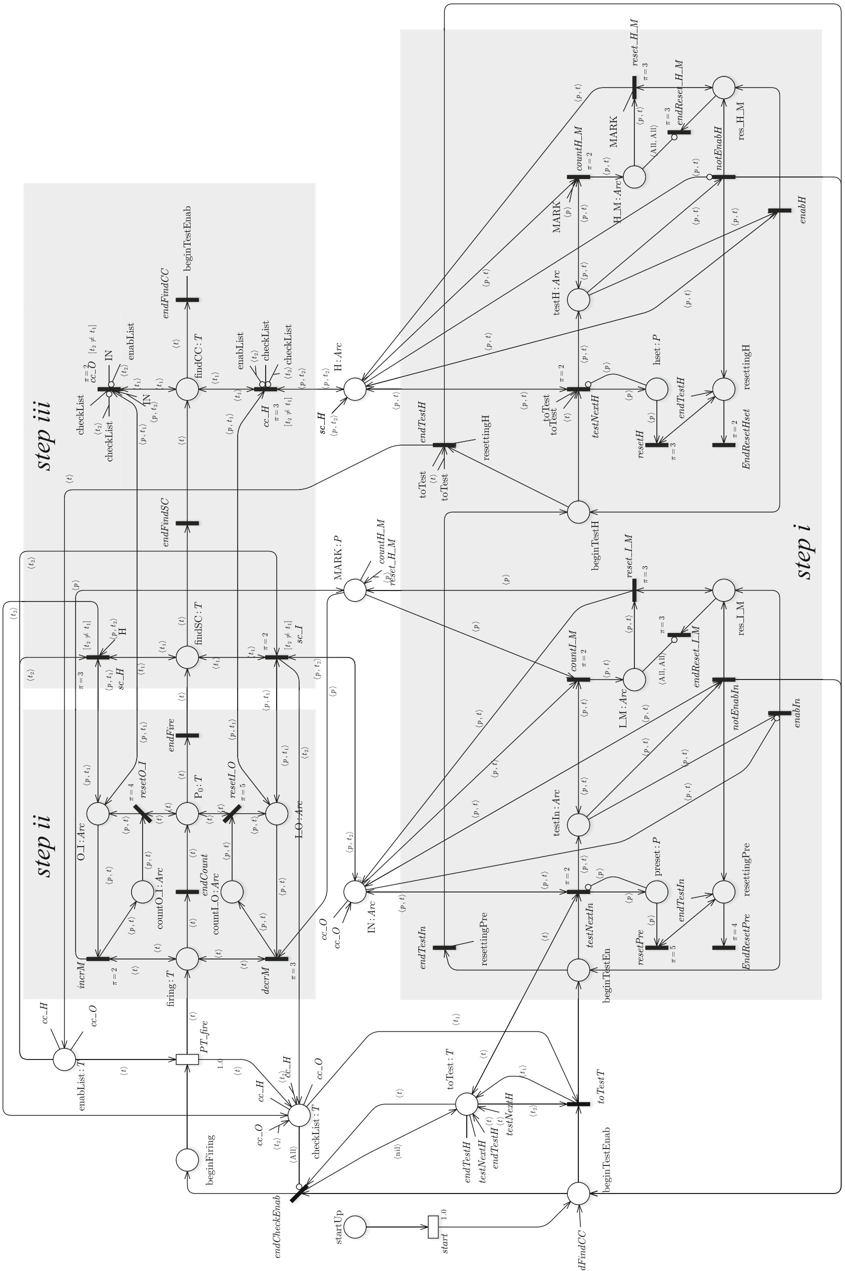
In this paper, we introduce an SN-based formal modeling framework for self-adaptive systems with a decentralized adaptation control. SNs are a high-level PN formalism with a syntax that outlines behavioral symmetries of systems. The framework builds on an SN component that emulates the behavior of any P/T system (with inhibitor arcs) encoded as an SN marking. The P/T system represents the base-level of the model, which can be arbitrarily adapted using transformation rules/procedures logically forming the model's high-level layer. These two components are coherently linked through a net-interface.

Our approach exploits consolidated analysis techniques implemented by off-the-shelf software tools. We provide a detailed, formal presentation of both the SN-based emulator and the evolutionary components, that include an API of base transformations primitives on which distributed adaptation procedures can be specified. Symbolic structural analysis techniques have been used to validate the components of the emulation-based framework. State-space exploration techniques have been used instead to formally verify the modeled systems. At this stage, all the native SN features can be used: generation of a quotient (symbolic) state space (exploiting behavioral symmetries), timed analysis (thanks to the native stochastic extension of SN), and model checking to extract meaningful insights from both the base-level layer (i.e., the managed subsystem) and the high-level layer (i.e., the managing subsystem). The usability and effectiveness of our approach have been shown on two variants of a self-healing manufacturing system. Our experiments show both the applicability and the cost (in terms of time and space) required by verification activities. Major complexity concerns have been illustrated, discussed, and (partially) faced. All the models introduced in this paper have been released publicly to encourage the repetition of experiments.

Our ongoing research activity includes the following main directions. We are developing an easy-to-use Domain Specific Language to help non-expert users specify the decentralized adaptation control loops and the (arbitrarily complex) patterns of coordination among them. Furthermore, we are going to include explicit modeling of uncertain aspects of the surroundings employing probabilistic approaches, as suggested by Camilli et al. (2017b, 2018). We also aim at addressing the scalability and complexity concerns of the approach that arise when modeling realistic systems. On the one hand, we plan to enhance the current version of the symbolic state-space builder of GreatSPN by exploiting symbolic structural analysis to reduce the immediate transitions' interleaving. Indeed, the huge number of immediate transitions needed to preserve the atomic firing semantics of the base-level and to ensure the coherence in adaptation steps has been identified as a major bottleneck of our approach. We also believe that simple syntactical extensions of SN, such as flush-arcs and symmetric marking-dependent guards, could greatly alleviate this problem. On the other one hand, we aim at exploiting the modularity of the model and some invariant properties in its fixed parts.

Appendix A: Emulator SN model

Figure 16 shows the whole SN model implementing our emulating framework described in Section 5. This model has been obtained mechanically by translating the listing reported in Algorithm 1. The marking encodes the AMS case study described in Section 3.



class $P = \{\texttt{pin}_1\}$ is $\texttt{Input} + \texttt{pl}\{1..2\}$ is $\texttt{Loaded} + \texttt{pw}\{1..2\}$ is $\texttt{Worked} +$ $\texttt{pb}\{1..2\}$ is $\texttt{Broken} + \{\texttt{pa}_1\}$ is $\texttt{Assembled} + \{\texttt{pnf}_1\}$ is $\texttt{NoFault}$

class $T = \{\texttt{tlo}_1\}$ is $\texttt{Loader} + \texttt{tli}\{1..2\}$ is $\texttt{Line} + \{\texttt{pa}_1\}$ is $\texttt{Assembler} +$ $\texttt{tfa}\{1..2\}$ is $\texttt{Fault} + \{\texttt{trs}_1\}$ is $\texttt{Restart} + \{\texttt{tfl}_1\}$ is $\texttt{Flush} +$ $\{\texttt{tre}_1\}$ is $\texttt{Repair} + \{\texttt{null}\}$ is $\texttt{nil}$

domain $Arc = P \times T$

Fig. 16 Symmetric net emulating model along with color classes/domains

Funding Open access funding provided by Libera Università di Bolzano within the CRUI-CARE Agreement.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommonshorg/licenses/by/4.0/.

References

Allen R, Douence R, Garlan D (1998) Specifying and analyzing dynamic software architectures. In: FASE, pp 21–37. https://doi.org/10.1007/BFb0053581

Amparore EG, Donatelli S (2010) MC4CSL^{TA}: An efficient model checking tool for cslta. In: 2010 Seventh International Conference on the Quantitative Evaluation of Systems, pp 153–154

Arcaini P, Riccobene E, Scandurra P (2017) Formal design and verification of self-adaptive systems with decentralized control. ACM Trans Auton Adapt Syst 11(4):25:1–25:35. https://doi.org/10.1145/3019598

Baarir S, Beccuti M, Cerotti D, Pierro MD, Donatelli S, Franceschinis G (2009) The GreatSPN tool: Recent enhancements. SIGMETRICS Perform Eval Rev 36(4):4–9. https://doi.org/10.1145/1530873.1530876

Balbo G (2001) Introduction to stochastic Petri nets. In: Brinksma E, Hermanns H, Katoen J-P (eds) Lectures on Formal Methods and PerformanceAnalysis: First EEF/Euro Summer School on Trends in Computer Science Bergen Dal, The Netherlands, July 3–7, 2000 Revised Lectures. Springer, Berlin, pp 84–155

Bartels B, Kleine M (2011) A CSP-based framework for the specification, verification, and implementation of adaptive systems. In: Giese H, Cheng BHC (eds) 2011 ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011. ACM, Waikiki, pp 158–167. https://doi.org/10.1145/1988008.1988030

Bellettini C, Camilli M, Capra L, Monga M (2012) Symbolic state space exploration of rt systems in the cloud. In: 2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, pp 295–302

Brun Y, Marzo Serugendo G, Gacek C, Giese H, Kienle H, Litoiu M, Müller H, Pezzè M, Shaw M (2009) Software engineering for self-adaptive systems. In: Cheng BH, Lemos R, Giese H, Inverardi P, Magee J (eds). Springer, Berlin, pp 48–70. https://doi.org/10.1007/978-3-642-02161-9_3

Cabac L, Duvigneau M, Moldt D, Rölke H (2005) Modeling dynamic architectures using nets-within-nets. In: Proceedings of the 26th International Conference on Applications and Theory of Petri Nets. Springer, Berlin, pp 148–167

Camilli M, Bellettini C, Gargantini A, Scandurra P (2018) Online model-based testing under uncertainty. In: 2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE), pp 36–46

Camilli M (2014) Formal verification problems in a big data world: towards a mighty synergy. In: Jalote P, Briand LC, van der Hoek A (eds) 36th International Conference on Software Engineering, ICSE '14, Companion Proceedings, Hyderabad, India, May 31 - June 07, 2014. ACM, pp 638–641. https://doi.org/10.1145/2591062.2591088

Camilli M, Bellettini C, Capra L (2018) A high-level Petri net-based formal model of distributed self-adaptive systems. In: Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings, ECSA'18. Association for Computing Machinery, New York. https://doi.org/10.1145/3241403.3241445

Camilli M, Capra L, Bellettini C (2019) Pnemu: An extensible modeling library for adaptable distributed systems. In: Donatelli S, Haar S (eds) Application and Theory of Petri Nets and Concurrency. Springer International Publishing, Cham, pp 80–90

Camilli M, Gargantini A, Scandurra P (2015) Specifying and verifying real-time self-adaptive systems. In: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), pp 303–313

Camilli M, Gargantini A, Scandurra P (2018) Zone-based formal specification and timing analysis of real-time self-adaptive systems. Sci Comput Program 159:28–57. https://doi.org/10.1016/j.scico.2018.03.002

Camilli M, Gargantini A, Scandurra P, Bellettini C (2017a) Event-based runtime verification of temporal properties using time basic Petri nets. In: Barrett C, Davies M, Kahsai T (eds) NASA formal methods. Springer International Publishing, Cham, pp 115–130. ISBN 978-3-319-57288-8

Camilli M, Gargantini A, Scandurra P, Bellettini C (2017b) Towards inverse uncertainty quantification in software development (short paper). In: Cimatti A, Sirjani M (eds) Software Engineering and Formal Methods - 15th International Conference, SEFM 2017, Proceedings, Lecture Notes in Computer Science, vol 10469. Springer, Trento, pp 375–381. https://doi.org/10.1007/978-3-319-66197-1_24

Capra L, De Pierro M, Franceschinis G (2015) Computing structural properties of symmetric nets. In: Quantitative Evaluation of Systems : 12th International Conference, QEST 2015, Proceedings, vol 9259. Springer International Publishing, Madrid, pp 125–140

Capra L, Pierro MD, Franceschinis G (2005) A High Level Language for Structural Relations in Well-Formed Nets. In: Proceedings of the 26th Int. Conf. ATPN 2005, LNCS 3536. Springer, pp 168–187. https://doi.org/10.1007/11494744_11

Capra L (2016) A pure spec-inscribed pn model for reconfigurable systems. In: 2016 13th International Workshop on Discrete Event Systems (WODES). IEEE Computer Society, pp 459–465

Capra L, Camilli M (2018) Towards evolving Petri nets: a symmetric nets-based framework. IFAC-PapersOnLine 51(7):480–485. https://doi.org/10.1016/j.ifacol.2018.06.343. 14th IFAC Workshop on Discrete Event Systems WODES 2018

Capra L, Camilli M (2020) Emulating self-adaptive stochastic Petri nets. In: Gribaudo M, Iacono M, Phung-Duc T, Razumchik R (eds) Computer Performance Engineering. Springer International Publishing, Cham, pp 33–49

Capra L, Cazzola W (2006) A Petri-net based reflective framework for the evolution of dynamic systems. Electron Notes Theor Comput Sci 159:41–59

Chiola G, Dutheillet C, Franceschinis G, Haddad S (1993) Stochastic well-formed colored nets and symmetric modeling applications. IEEE Trans Comput 42(11):1343–1360. https://doi.org/10.1109/12.247838

Chiola G, Marsan MA, Balbo G, Conte G (1993) Generalized stochastic Petri nets: A definition at the net level and its implications. IEEE Trans Softw Eng 19:89–107

de Lemos R, Garlan D, Ghezzi C, Giese H (eds) (2017) Software engineering for self-adaptive systems III. assurances - international seminar, revised selected and invited papers, Lecture Notes in Computer Science, vol 9640. Springer, Dagstuhl Castle. https://doi.org/10.1007/978-3-319-74183-3

de Lemos R, Giese H, Müller HA, Shaw M (eds) (2013) Software engineering for self-adaptive systems II - international seminar, 2010 revised selected and invited paper, Lecture Notes in Computer Science, vol 7475. Springer, Dagstuhl Castle. https://doi.org/10.1007/978-3-642- 35813-5

de Lemos R, Giese H, Müller HA, Shaw M, Andersson J, Litoiu M, Schmerl B, Tamura G, Villegas NM, Vogel T, Weyns D, Baresi L, Becker B, Bencomo N, Brun Y, Cukic B, Desmarais R, Dustdar S, Engels G, Geihs K, Göschka KM, Gorla A, Grassi V, Inverardi P, Karsai G, Kramer J, Lopes A, Magee J, Malek S, Mankovskii S, Mirandola R, Mylopoulos J, Nierstrasz O, Pezzè M, Prehofer C, Schäfer W, Schlichting R, Smith DB, Sousa JP, Tahvildari L, Wong K, Wuttke J (2013) Software engineering for self-adaptive systems: A second research roadmap. In: de Lemos R, Giese H, Müller HA, Shaw M (eds) Software Engineering for Self-Adaptive Systems II: International Seminar, 2010 Revised Selected and Invited Papers. Springer, Berlin, pp 1–32. https://doi.org/10.1007/978-3-642-35813-5_1

Dicesare F, Harhalakis G, Proth J-M, Silva M, Vernadat F (1993) Practice of Petri nets in manufacturing, vol 45. Springer, Dordrecht

Gagniuc PA (2017) Markov chains: From theory to implementation and experimentation. In: Markov Chains. Wiley

Goguen J, Kirchner C, Meseguer J, Kirchner H, Winkler T, Megrelis A (1988) An introduction to obj 3. In: 1st International Workshop on Conditional Term Rewriting Systems. Springer, London, pp 258–263

Hoffmann K, Mossakowski T (2002) Algebraic higher-order nets: Graphs and Petri nets as tokens. In: Wirsing M, Pattinson D, Hennicker R (eds) Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Revised Selected Papers, LNCS, vol 2755. Springer, Frauenchiemsee, pp 253–267

Hoffmann K, Ehrig H, Mossakowski T (2005a) High-level nets with nets and rules as tokens. In: Proceedings of the 26th International Conference on Applications and Theory of Petri Nets, ICATPN'05. Springer, Berlin, pp 268–288

Jensen K (1997) Coloured Petri nets. basic concepts, analysis methods and practical use, vol 1, Basic Concepts. Monographs in Theoretical Computer Science, Springer, 2nd corrected printing 1997. ISBN: 3-540-60943-1

Köhler-Bußmeier M (2009) Hornets: Nets within nets combined with net algebra. In: Franceschinis G, Wolf K (eds) Applications and Theory of Petri Nets. Springer, Berlin, pp 243–262

Li J, Dai X, Meng Z, Dou J, Guan X (2009) Rapid design and reconfiguration of Petri net models for reconfigurable manufacturing cells with improved net rewriting systems and activity diagrams. Comput Ind Eng 57(4):1431–1451. https://doi.org/10.1016/j.cie.2009.07.013, http://www.sciencedirect.com/science/article/pii/S0360835209002137

Llorens M, Oliver J (2004) Structural and dynamic changes in concurrent systems: reconfigurable Petri nets. IEEE Trans Comput 53(9):1147–1158. https://doi.org/10.1109/TC.2004.66

Meng X (2010) Modeling of reconfigurable manufacturing systems based on colored timed object-oriented Petri nets. J Manuf Syst 29(2):81–90. https://doi.org/10.1016/j.jmsy.2010.11.002

Reisig W (1985) Petri nets: An introduction. Springer, New York

Reisig W (1991) Petri nets and algebraic specifications. Theor Comput Sci 80(1):1–34

Salehie M, Tahvildari L (2009) Self-adaptive software: Landscape and research challenges. ACM Trans Auton Adapt Syst 4(2):14:1–14:42. https://doi.org/10.1145/1516533.1516538

Valk R (1998) Petri nets as token objects: An introduction to elementary object nets. In: Proceedings of the 19th International Conference on Application and Theory of Petri Nets, ICATPN '98. Springer, London, pp 1–25

Valk R (2004) Object Petri nets. In: Desel J, Reisig W, Rozenberg G (eds) Lectures on Concurrency and Petri Nets: Advances in Petri Nets. Springer, Berlin, pp 819–848

Weyns D, Iftikhar MU, de la Iglesia DG, Ahmad T (2012) A survey of formal methods in self-adaptive systems. In: Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering, C3S2E '12. ACM, New York, pp 67–79. https://doi.org/10.1145/2347583.2347592

Weyns D, Schmerl B, Grassi V, Malek S, Mirandola R, Prehofer C, Wuttke J, Andersson J, Giese H, Göschka KM (2013) On patterns for decentralized control in self-adaptive systems. In: de Lemos R, Giese H, Müller HA, Shaw M (eds) Software Engineering for Self-Adaptive Systems II: International Seminar, 2010 Revised Selected and Invited Papers. Springer, Berlin, pp 76–107. https://doi.org/10.1007/978-3-642-35813-5_4

Zhang J, Cheng BHC (2006) Model-based development of dynamically adaptive software. In: Proceedings of the 28th International Conference on Software Engineering, ICSE '06. ACM, New York, pp 371–380. https://doi.org/10.1145/1134285.1134337

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Matteo Camilli is currently a junior assistant professor at the Faculty of Computer Science of the Free University of Bozen-Bolzano (Italy). He received the Ph.D. degree in Computer Science from the University of Milan (Italy) in 2015. His current research activity focuses on formal methods and software engineering. He is especially interested in: software requirements specification, analysis and verification; model-based testing; models at runtime; uncertainty mitigation; risk management; design-time and runtime verification of software systems; formal modeling and simulation.

Lorenzo Capra is an assistant professor at the Computer Science dept. of the University of Milan (Italy). He received the Ph.D. degree in Computer Science from the University of Turin (Italy) in 2001. He is especially interested in structural analysis techniques and tools for High-level Petri nets based on algebraic approaches, and in modeling dynamic discrete-event systems.