Theoretical Computer Science 990 (2024) 114397

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Modular rewritable Petri nets: An efficient model for dynamic distributed systems ☆,☆☆

Lorenzo Capra [a,*], Michael Köhler-Bußmeier [b]

[a] *Università degli Studi di Milano, Via Celoria 18, Milano, 20133, Italy*
[b] *University of Applied Science Hamburg, Berliner Tor 7, Hamburg, D-20099, Germany*

ARTICLE INFO

Keywords:
Maude
PT nets
Distributed systems
Adaptation
Model-checking

ABSTRACT

Modern distributed systems are becoming pervasive and increasingly provided with adaptation, (self-)reconfiguration and mobility capability. On one side, to face the challenges of the highly dynamic environments where they are deployed. On the other side, to keep production/maintenance costs down. Therein lies the increasing demand for formal models encompassing all of these aspects (besides concurrency). Hardly any of the classical formalisms like Petri Nets, Automata, and Process Algebra, even though powerful, permits designers to easily specify dynamic structural changes to systems and evaluate their impact on system behaviour. That has led to several extensions of classical formal models (e.g., the Pi calculus or the Nets-within-Nets paradigm) rarely accompanied by suitable analysis techniques. A recent formalization of a class of Rewritable Place-Transition Nets (RwPT) in Maude has proved potentially convenient to specify dynamically reconfigurable systems. Concerning analogous proposals, the RwPT formalism provides more abstraction/flexibility in modelling and efficiency in analysis. Nevertheless, its ability to scale the size of distributed systems built of several similar (nested) components is limited.

This paper presents a compositional approach to define large RwPT models in a typical algebraic way and to exploit the modular structure of models during the analysis: Symmetries are implicitly captured by composite node-labelling (reflecting the model's hierarchical structure) that is preserved by net rewrites. A distributed, gracefully degrading production system is used as a case study. Experimental evidence points out the dramatic impact of the approach against a non-modular one and the advantages over alternative techniques. Even if the emphasis is on state-space-based verification, the paper shows the convenience of combining it with structural analysis, which is typical of Petri nets as well. For that purpose, rewrite-rule abstractions are given in the form of guidelines.

☆ This article belongs to Section A: Algorithms, automata, complexity and games, Edited by Paul Spirakis.
☆☆ Extended version for TCS.

* Corresponding author.
E-mail addresses: capra@di.unimi.it (L. Capra), michael.koehler-bussmeier@haw-hamburg.de (M. Köhler-Bußmeier).

https://doi.org/10.1016/j.tcs.2024.114397
Received 9 September 2023; Accepted 10 January 2024

Available online 17 January 2024

0304-3975/Â© 2024 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

Modern distributed systems operate with varying conditions in highly dynamic environments. System components may become temporarily or permanently unavailable, may appear/disappear, e.g., due to failures or dynamic load-balancing. The ability to structurally reconfigure is increasingly adopted to face such complexity. Self-adaptive systems (and other types, e.g., automated or embedded) rely on dynamic reconfiguration strategies overlapping with the system base functionality.

There is an impelling need for efficient formal methods covering the system base functionality and reconfiguration issues to validate early design choices or verify the system behaviour at run-time.

Petri nets (PN) are a central model for concurrent or distributed systems. Classical PNs, however, are not expressive enough to easily specify dynamically reconfigurable systems. Several PN extensions have been proposed in which enhanced expressivity is not adequately supported by analysis techniques and tools. The most significant representatives belong to the "Nets-within-Nets" paradigm introduced in [52] and consist of High-Level PNs (PNs with individual tokens) enriched with algebraic net-operators, e.g., [31,11,26]. The theory of Nets-within-Nets gives clear evidence that adaptive/reconfigurable systems need an ad-hoc formalism: A base property like reachability, indeed, requires at least exponential space for bounded systems in the case of Nets-within-Nets [33,35,32], while this problem is PSpace-complete for Place-Transition (PT) nets. Namely, to mimic the behaviour of a self-adaptive system based on Nets-within-Nets we should construct an exponentially larger PT net.

As for PNs with anonymous tokens, we have to mention Reconfigurable PN, a family of formalisms composed of a marked net and a set of rewrite rules specified as *pushouts*, according to algebraic Graph Transformations Systems [37,21,30,47]. Research in this field was principally foundational, aiming to rephrase these models as $\mathcal{M}$-adhesive categories. We refer to [45] for a survey.

In [14] we have formalized a type of "rewritable" PT nets (equipped with inhibitor arcs, i.e., Turing-complete by themselves) in `Maude` [19], a purely declarative language with intuitive operation semantics and sound denotational semantics in rewriting-logic [40,8]. The point of view of [14], and this paper as well, is principally operational: The intent is to propose a scalable `Maude`-based framework for the validation and analysis of distributed systems formally specified as PT nets structurally evolving.

Compared to related proposals [3,46], where a simpler class of PNs is translated in `Maude` to exploit the associated model-checking facilities, the Rewritable PT formalism defined in [14] (RwPT for short) provides more data abstraction (to ease the modeller task), is more compact and then efficient, and removes the constraints imposed by the adoption of "pushout" as rule style [3,46].

RwPT is an instance of Graph Transformation Systems. In this context, we have to reason up to graph isomorphism (GI): Recognizing (and folding) equivalent states reached during model dynamics is essential to scale model size or parallelism level. In particular, when we associate a stochastic process (usually a Markov Chain) with the model state transition system. Another aspect is the formal verification of properties (usually) through model-checking facilities: Writing or checking formulae matching hundreds or thousands of (equivalent) states is far from trivial. Despite the numerous studies, some uncertainty on GI's complexity remains: it belongs to NP, but it is thought neither P nor NP-complete. Recent work [2] seems to indicate that GI is quasi-polynomial. Graph Canonization (GC) is an approach which consists in finding for any G a *canonized* representative such that for any two G, G' $G \simeq G' \Leftrightarrow can(G) = can(G')$. GC is at least as complex as GI. Despite its algorithmic overhead, however, it has many advantages. Since the canonized form of G is unique we compute it once and for all: Afterwards, we directly compare the canonized forms of candidate isomorphic graphs. GC seems a natural approach in the `Maude` setting.

We have defined a PT canonization technique [13] (fully integrated into `Maude`) to use in the RwPT context, inspired by existing algorithms on graphs. This technique has proved effective –in terms of space reduction– and relatively efficient –in terms of execution time– when applied to models with an irregular/asymmetric layout. Conversely, it has proved highly inefficient when applied to more realistic models with several similarly behaving distributed components/modules glued together somehow. Therefore, the approach presented here puts attention to symmetries.

This paper significantly extends [14] by facing rewritable PT (RwPT) scalability in both model construction and (reachability) analysis. We define a base set of algebraic net operators that allow for the automatic detection of behavioural equivalences during state space construction. These operators incrementally annotate PT nodes (places) using a modular/hierarchical labelling that reflects the model symmetry structure. Through these composite annotations, we can bring PT system terms to a *normal* form very efficiently. For that, we define a new PT encoding even simpler than [14].

We use as a benchmark the model of a distributed, gracefully degrading production system. This model is a significant extension of the one employed in [14]. We present a series of experiments (conducted on standard hardware) showing the effectiveness of the compositional approach in terms of time and space and the ability to scale adequately the model size. This part includes a comparison with alternative models based (in particular) on Symmetric Nets, a standard Coloured PN formalism whose syntax implicitly captures model symmetries.

Despite focusing on state space analysis (model-checking), we show the advantages of a PN-based formalization by briefly discussing its fruitful integration with structural techniques. In particular, we outline rewrite-rule abstractions to (over)estimate the topology of dynamic distributed systems.

The contribution has the following structure: Section 2 recalls basic definitions for PT nets and describes the basic components of our running example of a distributed (fault-tolerant) production system. Section 3 gives a short introduction into Rewriting Logic and the `Maude` system. Section 4 presents the encoding of Rewritable PT in `Maude`. Section 5 demonstrates the viability of formal verification with Rewritable PT nets. Then, Section 6 exploits symmetry of net operators to face the complexity of large-scale models. This part is the main contribution of this extended version, together with Section 7, which presents experimental evidence. Section 8 describes a state abstraction technique. Section 9 completes the (non-exhaustive) survey of related works. Section 10 outlines the

weak and strong points of rewritable PT integrated with compositional operators. The paper ends with a conclusion and ongoing work.

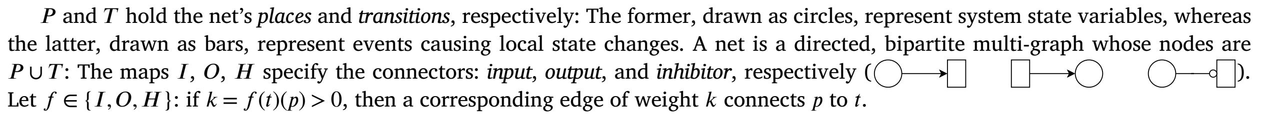
2. PT nets and systems

This section contains the definition of (Turing-powerful) PT nets and a description of the main PT components used in the rest of the paper.

2.1. (T-labelled) place/transition nets with inhibitor arcs (PT nets)

Multisets A *multiset* (*bag*) b on a set D is a map $b : D \to \mathbb{N}$, where $b(d)$ is the *multiplicity* of d. We say d is an element of b ($d \in b$) if and only if $b(d) > 0$. Bag operators are defined component-wise. Let $Bag[D]$ be the set of bags on D, and $b_1, b_2 \in Bag[D]$: $b_1 + b_2$ and $b_1 - b_2$, both in $Bag[D]$, are defined $\forall d \in D$: $b_1 + b_2(d) = b_1(d) + b_2(d)$; $b_1 - b_2(d) = b_1(d) - b_2(d)$ if $b_1(d) \geq b_2(d)$, 0 otherwise. Similarly, $b_1 < b_2$ holds *true* if and only if $\forall d \in D$ $b_1(d) < b_2(d)$. The other operators are defined likewise.

A (T-labelled) PT *net* [48] is a 6-tuple (P, T, I, O, H, Λ), where: P, T are non-empty, finite, disjoint sets, I, O, H are maps: $T \to Bag[P]$ and $\Lambda : T \to L$ is a labelling function.

P and T hold the net's *places* and *transitions*, respectively: The former, drawn as circles, represent system state variables, whereas the latter, drawn as bars, represent events causing local state changes. A net is a directed, bipartite multi-graph whose nodes are $P \cup T$: The maps I, O, H specify the connectors: *input*, *output*, and *inhibitor*, respectively (). Let $f \in \{I, O, H\}$: if $k = f(t)(p) > 0$, then a corresponding edge of weight k connects p to t.

A net's *marking* is a bag $m \in Bag[P]$, which formally describes a distributed state. The *firing rule* specifies the PT dynamics. A transition $t \in T$ is *enabled* in a marking m if and only if:

$$I(t) \leq m \wedge H(t) > m$$

'>' being restricted to the elements of $H(t)$. A transition t which is enabled in m and such that $I(t) \neq O(t)$ can *fire* leading to m':

$$m' = m + O(t) - I(t)$$

The notation $m[t\rangle m'$ means that t is enabled m and leads to m' when fires.

A PT-*system* is a pair (N, m), where N is a net and m is a marking of N. The PT-system interleaving semantics is specified by the *reachability graph* (RG), an edge-labelled, directed graph (V, E) whose nodes are reachable markings. $RG(N, m_0)$ is defined inductively:

$$m_0 \in V;\ \text{if } m \in V \text{ and } m[t\rangle m' \text{ then } m' \in V \text{ and } m \xrightarrow{t} m' \in E.$$

T-labelling Λ associates extra or meta information to transitions not influencing the PT semantics. For example, in stochastic PT nets transitions fire with delays sampled from negative exponential distributions: in that case, $L = \mathbb{R}^+$. In this paper's context, L contains descriptive tags (strings).

2.2. Running example's building blocks

The two PT systems in Fig. 1 are the main components of the paper's running example (Section 4.4), a gracefully degrading, distributed production system. Fig. 1, with the addition of system evolution's description, is pretty similar to the benchmark used in [14] for RwPT and in [12] for a Symmetric Net emulator of dynamic PT systems.

The net at the top represents a production unit (PL in the rest) nominally composed of K production lines (robots) that work a K multiple of raw pieces: These lines (the sub-nets $\{w_i, ln_i, a_i\}$, $i : 0 \ldots K-1$) are symmetric, i.e., fully interchangeable. An assembler (transition as) brings together K worked pieces into an artefact. A loader (transition ld) picks up K pieces at a time from a warehouse (place s) onto the lines. In our case study, $K = 2$. A parameter $M \in \mathbb{N}^+$ defines the number of raw pieces worked during a production cycle: The marking of s is initially $K \cdot M$. To ensure a cyclic behaviour, each artefact out is balanced by K raw pieces in.

A line occasionally gets faulty, so the production unit adapts to continue working only with the left line(s). A fault occurrence (transitions ft_i) causes a line's breakdown, modelled by an inhibitor arc. Simultaneous faults have a null probability (ft_0 and ft_1 are mutually in conflict). Using simple static analysis (Section 4.4), we can prove that the system enters a *deadlock* following a line's breakdown, with half (raw) pieces (M) on the broken line and half (worked, ready to be assembled) on the left line.

The net at the bottom of Fig. 1 shows the PL evolution following a deadlock: Besides a gradual change to the PL layout, adaptation causes moving pieces from the broken line to the other(s) to preserve the production cycle. Pieces left on the broken line (place w_0 in the example) are transferred to the other line (place w_1): The marking of the PT net at the bottom describes the state following adaptation. Differently from [14] and [12], we assume that a production unit cannot recover from a second fault affecting it (Section 4.4). Instead, we will consider a more complex scenario with many PL replicas cooperating and implementing a distributed gracefully degrading system.

The limits in the expressivity of PT nets make the formalization of a dynamic scenario like that described virtually unfeasible.

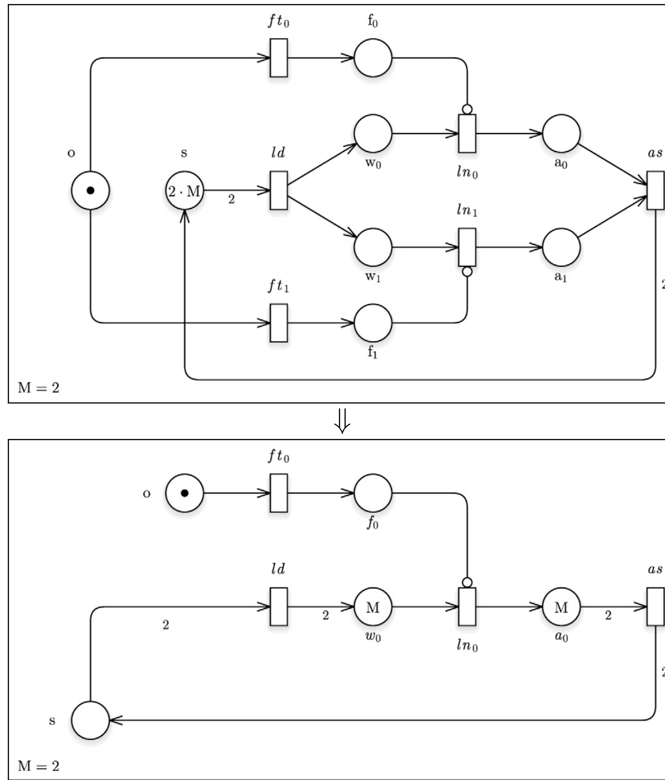


Fig. 1. Single Production Unit (PL) and adaptation following a fault.

3. Rewriting logic and the `Maude` system

`Maude` [19] is an expressive, purely declarative language with a rewriting logic semantics [40,8]. `Maude`'s statements are (conditional) *equations* (keyword `eq`) and *rules* (keyword `rl`). Both sides of a rule/equation are terms of a given *kind* that may hold variables. Rules and equations have simple rewriting semantics in which instances of the left-hand side are replaced by corresponding instances of the right-hand side.

A `Maude` *functional* module (keyword `fmod`) contains only equations and corresponds to a functional program defining one or more operations through equations used as simplifications. A `Maude` *system module* (keyword `mod`) contains rules and possibly equations. Rules are similarly computed by rewriting terms from left to right but represent local transitions in a (concurrent) system. Although declarative in style and with sound logical semantics, system modules are non-functional. In `Maude`, a state of a distributed system is usually represented as an associative "multi-set": Rules apply *concurrently* to different portions of the multi-set, leading to a new state. There is no assumption on the confluence of rewrites.

`Maude` features expressivity, simplicity and performance. A wide range of systems is expressible with `Maude`, which may be used as a specification language, a programming language and a meta-language (in which other languages and logics can be expressed). `Maude` achieves expressivity through equational pattern matching modulo operator equational attributes; user-definable operator syntax and evaluation strategy; sub-typing and partiality; generic types; reflection. A `Maude` program is a logical theory, and a `Maude` computation is a deduction according to the theory axioms. Under some executability conditions, the mathematical and the operational semantics coincide ([6,54]).

A functional module specifies an *equational theory* in membership equational logic [41,6]. Formally, such a theory is a pair $(\Sigma, E \cup A)$, where Σ is the signature, that is, the specification of all the sorts, subsorts, kinds,[1] and (overloaded) operator declarations (considering also the imported functional modules); E is the collection of (conditional) equations and memberships declared in the module(s), and A is the collection of equational attributes (`assoc`, `comm`, and so on) of operators (treated as predefined equations). The models of functional modules are algebras, namely, data sets with related operations. The family of Σ-ground terms T_Σ defines a model called Σ-algebra ($T_{\Sigma(X)}$ denotes the whole set of terms). According to [10], the best model of $(\Sigma, E \cup A)$ satisfies $E \cup A$ and is both junk-free (all the elements are denoted by T_Σ terms) and confusion-free (only those elements forced to be equal by $E \cup A$ are identified). This model, which is called the *initial algebra* of $E \cup A$ and is denoted $T_{\Sigma/E \cup A}$, exists [6] and provides the denotational

[1] A *kind* is an implicit equivalence class gathering all sorts connected by subsort relation; terms having a kind but not a sort may be considered as *undefined* or *errors*.

semantics of the functional module specifying $(\Sigma, E \cup A)$. $T_{\Sigma/E\cup A}$ is the quotient of T_Σ in which the equivalence classes hold terms that prove equal using $E \cup A$.

If the axioms in E are Church-Rosser and terminating modulo A (each ground term is thus simplified uniquely regardless of the order in which equations apply) there is an intuitive, equivalent description for $T_{\Sigma/E\cup A}$. The final values (*canonical* forms) of all ground terms form an algebra called the *canonical term algebra*, denoted $CAN_{\Sigma/E\cup A}$. The `reduce` command of `Maude` interpreter reduces operators to their final values. The fact that the denotational and operational semantics coincide is expressed by $T_{\Sigma/E\cup A} \cong CAN_{\Sigma/E\cup A}$.

A `Maude` system module, with the imported submodules, specifies a generalized *rewrite theory* [40,8], that is, a four-tuple $\mathcal{R} = (\Sigma, E \cup A, \phi, R)$ where $(\Sigma, E \cup A)$ is the membership equational theory specified by the signature, equational attributes, and equation statements in the module; ϕ is a map specifying, for each operator in Σ, its frozen arguments; and R is a set of rewrite rules.[2] Intuitively, a rewrite theory specifies a concurrent system. The equational part $(\Sigma, E \cup A)$ represents the algebraic structure of the states, formalized by the initial algebra $T_{\Sigma/E\cup A}$. The rules R (with ϕ) specify the system dynamics, namely, the concurrent transitions. In this context, they represent structural changes to a PT system.

In rewriting logic, concurrent transitions become rewrite proofs; since many execution proofs may correspond to one computation (because of equivalent interleavings), rewriting logic has an equational theory of proof equivalence [8,40]. The initial model $\mathcal{T}_\mathcal{R}$ of $\mathcal{R}$ associates to each kind k a labelled transition system (a category) whose states are $T_{\Sigma/E\cup A,k}$, and whose transitions take the form: $[t] \xrightarrow{[\alpha]} [t']$, with $[t],[t'] \in T_{\Sigma/E\cup A,k}$, and $[\alpha]$ an equivalence class of rewrites modulo the equational theory of proof-equivalence. Different $[\alpha]$ represent different "truly concurrent" computations.

The executability of system modules is ensured by the (ground) *coherence* between rules and equations [54]. Assuming that $E \cup A$ is Church-Rosser and terminating, an efficient strategy (adopted by `Maude rewrite` command) is first to apply the equations to get a canonical form, then a rule in R. Coherence ensures completeness, i.e., any rewrite of $t \in T_\Sigma$ with R is also possible with t's canonical form. Coherence reduces rewriting with R modulo $E \cup A$, which is generally undecidable, to rewriting with E and R modulo A, which is decidable given an A-matching algorithm.

4. Formalization of rewritable PT with Maude

This section presents a formalization of *modular* rewritable PT nets in `Maude`. A RwPT is a PT net –formally represented as an algebraic term– with associated rewrite rules including the firing rule and, possibly, net transformations. The `Maude` sources are available at github.com/lgcapra/RwPT/modPT.

Checking the confluence, termination and coherence of `Maude` modules is under the user's responsibility. It is often possible to prove these properties (which guarantee module executability) using the `Maude` Formal Environment https://github.com/maude-team/MFE, largely integrated into the `Maude`'s interpreter. We have proven that the modules mentioned in this paper are executable. The functional part meets two other desirable properties: i) each ground term has a *least* sort (or kind); ii) the canonical form of ground terms is built of *constructors* (operators marked with the `ctor` attribute). We have (semi-automatically) verified syntactical conditions (term pre-regularity and sufficient completeness) for properties i) and ii).

Compared to our former proposal [14], we emphasize model compositionality and analysis (model-checking) efficiency. The new RwPT definition, which inherits some base elements from [14], adopts structured annotations (that point out the symmetry present in the model's layout) and a *place-based* encoding more compact than the one employed in [14].

4.1. Module structure

The RwPT specification pervasively uses predefined generic types (the functional modules `MAP{X,Y}`, `LIST{X}`, `SET{X}`) and builds on `BAG{X}`: This module provides a convenient representation for multi-sets, not merely seen (e.g., in [51,46]) as free commutative monoids but as a rich data type[3]: The operators `_._`, `_+_`, `_[_]` `_-_`, `_<_`, `_>_`, `set` (the first two of which are *constructors*) provide all the needed abstraction. In particular, the commutative/associative `_+_` gives an intuitive description of a bag as a weighted sum, e.g., `3 . a + 1 . b`.

Listing 1 shows the `BAG` module, also as an illustrative example of a functional module. The signature (keywords `sort`, `subsorts`, `op`) and equations (keywords `eq`, `ceq`, `assoc`, `comm`, `id`) are recognizable. Advanced features like the `owise` equational attribute and `gather` operator evaluation strategy are used.

Table 1 summarizes the modules forming the RwPT specification, in (partial) order of dependency: A module may import others above it in the list, by preserving their initial semantics (if functional), possibly modulo renaming of sorts/operators. Some modules are generic, with the type parameter indicating the type of transition labels. The last module specifies the running example. The Appendix includes the core module `PT-NET` for the reader's convenience. Throughout the paper, we will enclose significant `Maude` excerpts, omitting the arity and range of operators when clear from the context.

[2] R rules do not apply to frozen arguments (not used in this context).

[3] *Theories* are abstract modules specifying the type-parameters of generic modules, *views* are used in module instantiations to map theories' sorts/operators to corresponding elements of concrete or even abstract modules. In this context, the elementary theory -which only requires a sort- and default views are mostly used.

Listing 1. The multiset specification.

```
fmod BAG{X :: TRIV} is
 protecting INT .
 protecting EXT-BOOL .
 sorts Bag{X} NeBag{X} ElBag{X} .
 subsorts ElBag{X} < NeBag{X} < Bag{X} .
 vars N M : NzNat .
 var K : Nat .
 vars X Y Z : X$Elt . *** the sort which maps to TRIV's sort
 vars B B' B'' : Bag{X} .
 vars NeB NeB' NeB'' : NeBag{X} .
 *** base constructors
 op nil : -> Bag{X} [ctor] .
 op _._ : NzNat X$Elt -> ElBag{X} [prec 35 ctor] .
 *** look up (elements' multiplicity)
 op _`[_`] : Bag{X} X$Elt -> Nat [prec 23] .
 eq (N . X + B)[X] = N .
 eq B[X] = 0 [owise] .
 *** sum (constructor)
 op _+_ : Bag{X} Bag{X} -> Bag{X} [prec 39 ctor assoc comm id: nil] .
 op _+_ : NeBag{X} Bag{X} -> NeBag{X} [ctor ditto] .
 eq N . X + M . X = (N + M) . X .
 *** difference (lower priority than sum)
 op _-_ : Bag{X} Bag{X} -> Bag{X} [prec 41 gather (E e) right id: nil] .
eq B + NeB - NeB = B . *** optimization
eq N . X + B - M . X + B' = if N > M then N - M . X + (B - B') else B - B' fi.
ceq B - NeB = B if B =/= NeB [owise] .
 *** set an element's multiplicity
 op set : Bag{X} X$Elt Nat -> Bag{X} .
 eq set(B, X, 0) = remove(B, X) .
 eq set(B + N . X, X, M) = B + M . X .
 eq set(B, X, M) = B + M . X [owise] .
 *** remove an element
 op remove : Bag{X} X$Elt -> Bag{X} .
 eq remove(B + N . X, X) = B .
 eq remove(B, X) = B [owise] .
 *** relational ops
 op _<=_ : Bag{X} Bag{X} -> Bool [prec 43] .
 ceq N . X + B <= B' = false if N > B'[X] .
 eq B <= B' = true [owise] .
 op _>_ : Bag{X} Bag{X} -> Bool [prec 43] . *** restricted version
 ceq N . X + B > B' = false if N <= B'[X] .
 eq B > B' = true [owise] .
 *** cardinality
 op |_| : Bag{X} -> Nat .
 op |_| : NeBag{X} -> NzNat .
 eq | B | = $bcard(B, 0) . *** tail recursion used for efficiency
 op $bcard : Bag{X} Nat -> Nat .
 eq $bcard(nil, K) = K .
 eq $bcard(N . X + B, K) = $bcard(B, K + N) .
endfm
```

Table 1
Modules of the RwPT specification.

Module	Type	Description
PLACE	f	PT places (and labels)
PBAG	f	multi-set of places (instance of BAG{X})
TMATRIX	f	transition incidence matrix
PT-NET{L}	f	PT net signature
PT-SYS{L}	f	PT system signature
NET-OP{L}	f	net-algebra operators
PT-NORM{L}	f	PT normalization operators
PT-EMU{L}	s	PT system plus firing rule
FT-PL	s	running example: FT production system

4.2. PT signature

The Maude PT signature is bag-oriented, as the definition given in Section 2: Terms of sort `Pbag` denote multi-sets of places. Nevertheless, it has some peculiarity: Places carry structured annotations. A place label (a term of sort `Lab`) is a non-empty list of elementary labels (terms of sort `Elab`), i.e., pairs of `String` and `Nat` terms. Each elementary label refers to a component in a group of similar inside a nested hierarchy: A label's suffix represents the root of the hierarchy. For example, `< "a" ; 0 > < "L" ; 1 >` labels the assembly place of line 1 of a production unit.[4] The idea is to provide compositional net operators that incrementally (and automatically) annotate places. We get `Place` terms using the constructor `op p : NeLab -> Place .` This way, places are in one-to-one correspondence with labels.

Net transitions are *implicit*: A transition's incidence matrix is a triplet of place multi-sets (the `[_,_,_]` constructor) representing its I, O, H connections. A net is a term of sort `Map{Imatrix,L}` (renamed `Net` for convenience), i.e., a set of entries

```
[I,O,H] |-> lab
```

in which a matrix has an associated label, whose type is the generic sort of module's type parameter. In the following, transition labels are simple descriptive tags (strings) used in net-algebra operators. In general, they may carry additional information such as metadata or timing.

A net's entry has sort `Transition`, a `Net`'s subsort (`Transition < Net`). Thus, using the associative/commutative `Net` juxtaposition (`_;_`) we may build nets in a pretty compositional way. For example, the sub-net made up of transitions ld, ln_0 in Fig. 1 (top) may be encoded as the `Net` term (`nilP` is the empty multi-set):

Listing 2. Maude's encoding of a subnet.

```
[2 . p(< "s" ; 0 >), 1 . p(< "w" ; 0 >) + 1 . p(< "w" ; 1 >), nilP] |-> "ld" ;
[1 . p(< "w" ; 0 >), 1 . p(< "a" ; 0 >), 1 . p(< "f" ; 0 > ] |-> "ln"
```

A PT system term (of sort `System`) is the empty juxtaposition (`__`) of a `Net` term and a `Pbag` term which represents a marking. For modelling reasons, we admit empty nets/systems (the terms `emptyN` and `emptyN nilP`, respectively).

Several operators allow for easy algebraic manipulation of PT nets, e.g.:

```
op enabled : Tmatrix Pbag -> Bool .
op dead : System -> Bool .
op dead : Tmatrix -> Bool .
op in :  Net Place -> Bool .
op clearup : System -> System .
```

The first two predicates implement the transition enabling and system deadlock condition, respectively; The third predicate checks for structurally dead transitions; The fourth predicate verifies memberships of a place to a net; The operator `clearup` removes redundant elements from a `System`: i) transitions t having null effect, namely, with $I(t) = O(t)$ ii) structurally dead transitions, such that $I(t)(p) \geq H(t)(p)$ for some p iii) marked places not appearing on the `Net` sub-term (we call such places *isolated*).

Properties The sketched formalization of PT nets, besides being suitable for modular model construction, has some advantages over the former one [14] based on an explicit representation of transitions and using rough node labelling:

- It doesn't require any consistency check (usually performed by adding membership equations) for `Net` and `System` terms.
- It is more efficient, in terms of rewriting (we have quantified a reduction of around 30%).
- It makes it possible to verify PT net automorphisms (i.e., symmetries) straightforwardly.

Conversely, it is a bit less intuitive, (apparently) unsuitable for a transition-based structural analysis and more wordy, due to richer annotations.

PT system dynamics The conditional rewrite rule `firing` specifies the PT firing rule. Apart from the use of a matching equation (`:=`)[5] to make the rule compact, the syntax looks similar to that used in Section 2.

[4] We use the empty concatenation `op __ : List{X} List{X}-> List{X}` and subsorting: `X$Elt < NeList{X} < List{X}`, i.e. (after renaming), `Elab < NeLab < Lab`.
In normalized terms, we use consecutive indices starting from zero.

[5] The rule's free variables I, O, H, are instantiated by matching the left-hand side of the matching equation against the canonical ground term bound to variable T.

Listing 3. The ordinary firing rule.

```
 vars I O H M : Pbag .
 var N : Net .
 var L : X$Elt . *** transition label
 var T : Tmatrix .
crl [firing] : (N ; T |−> L) M = > (N ; T |−> L) M + O − I if [I,O,H] := T /\ I =/= 0 /\ enabled(T, M) .
```

4.3. Rewritable PT nets

We assume that a Rewritable PT net is formalized in a system module (`FT-PL` in Table 1) which contains: i) two operators: `op net : -> Net.` and `op m0 : -> Pbag.` used as aliases ii) two equations that define their bindings to ground terms of sort `Net` and `Pbag`, respectively, iii) a set R of rewrite rules including `firing` (through module `PT-EMU` importation, see Table 1).

We may use the abbreviation (`net m0`, R) to denote a Rewritable PT net. R specifies the possible evolution of `net m0`: Besides the inner dynamics caused by transition firing, structural transformations triggered by conditions set on the system's current state/topology.

In related approaches [46], strongly inspired by (algebraic) Graph Transformation Systems, rewrite rules are *pushouts* and therefore have to meet non-trivial *gluing* conditions. Our formalization is at the same time more flexible and efficient, though rigorous.

We can classify rewrite rules based on the sort of terms in their left- and right-hand sides: `Place`, `Pbag`, `Transition`, `Net`, `System`. Except for `System` rules (the top-level ones), the others locally affect portions of `System` terms. This is fully coherent with the distributed operational semantics of rewriting logic. Nevertheless, it may give rise to undesired effects: For example, a rule of sort `Pbag` might (inadvertently) involve both the net marking and transition incidence matrix.

From a syntactical point of view, however, we potentially admit rules of *any* sort because they always rewrite a `System` term into another one. This is another advantage over [14].

As for the paper, we use top-level rules of type `System` and `Net`. Besides modelling reasons, this choice has a theoretical foundation: We bring `System` terms into pseudo-canonical forms (called *normalized*) that should not include unessential elements. Thus, we get rid of redundant elements (such as isolated places) –if any– by embedding suitable operators (e.g., `clearup`) in the rules' right-hand side.

Library of net-operators We have provided a library of base net operators (Listings 5, 6) with a twofold intent: To ease the modeller task in correctly specifying system adaptation; To permit the construction of large-scale models with several (nested) components by implicitly pointing out their symmetry.

Some basic tools are available in core modules, e.g., the `set` operator defined on bags allows one to change the weight of any PT edge and indirectly add/remove nodes to/from a net.

4.4. Base notions and properties of rewritable PT nets

In this section, we give a few intuitive notions and properties forming the theoretical background of rewritable PT nets. Unless otherwise specified, we refer to the *final* form of ground terms, that does exist and is well-defined (i.e., built of constructors and with a *least* sort) because all modules satisfy the executability conditions (cf. the end of Section 2).

Let $r:\ u \Rightarrow u'$ be a rewrite rule, with $u, u' \in T_{\Sigma(X),k}$. Let t, t' be ground terms of kind k and σ a ground substitution of r variables. The notation $t \overset{r(\sigma)}{\Rightarrow} t'$ means that t can be rewritten to t' via r and σ, namely, there exists σ: $\sigma(u) = t$ and $\sigma(u') = t'$.[6] In other words, σ provides a match for u against t. We may simply write $t \overset{r}{\Rightarrow} t'$.

The interleaving semantics of a rewritable PT net is (according to Rewriting Logic) a labelled transition system (TS) whose nodes are `System` ground terms. As usual, we define it inductively.

Definition 1 *(Semantics of rewritable PT).* Let (`net m0`, R) be a rewritable PT: The state-transition system TS(`net m0`, R) is an edge-labelled, directed graph (V, E) such that:

$$(\texttt{net m0}) \in V;\ \text{if } \texttt{s} \in V,\ r \in R \text{ and } \texttt{s} \overset{r(\sigma)}{\Rightarrow} \texttt{s'} \text{ then } \texttt{s'} \in V,\ \texttt{s} \overset{r(\sigma)}{\Rightarrow} \texttt{s'} \in E.$$

By default, the `Maude` inline `search` command explores the state-space starting from an initial term by applying rewrite rules one at a time in a fair, breadth-first way. Therefore, it is coherent with the definition above.

Property 1 *(PT systems are rewritable PT nets). A PT system (N, m_0) can be encoded as a term* (`net m0`) *such that*

$$RG(N, m_0) \cong TS(\texttt{net m0}, \{\texttt{firing}\})$$

(and vice-versa a non-empty `System` *term represents a PT-system).*

[6] In conditional rules, σ has to verify rule's condition. If u, u' are ground terms σ is empty.

Getting a `System` term out of a PT system (N, m) is straightforward: We only have to set an arbitrary bijection ψ_P between P and corresponding `Place` ground terms, which induces one between $Bag[P]$ and `Pbag` ground terms. Each transition $t \in T$ then maps to $[\psi_P(I(t)), \psi_P(O(t)), \psi_P(H(t))]$ `->` $\Lambda(t)$ (a `Transition` term), and N maps to the `Net` obtained by juxtaposing (using the associative `;`) the terms above. The isomorphism between RG and TS directly follows from the fact that, by definition, for any t and any marking m of N: $m[t\rangle m'$ if and only if $\psi_P(m) \overset{\texttt{firing}(\sigma)}{\longrightarrow} \psi_P(m')$, with $\sigma(\texttt{I}) = \psi_P(I(t)), \ldots, \sigma(\texttt{L}) = \Lambda(t)$ ($variables(\texttt{firing}) = \{\texttt{N,M,T,I,O,H,L}\}$). □

In rewritable PT nets, the firing rule has the same citizenship as the other rewrite rules. This makes formalization uniform and coherent with the concurrent rewriting logic semantics: If several rules match a `System` ground term, rewrites non-deterministically occur in an interleaved way. Conflicts may arise between PT transitions (matches of `firing` rule) and between (PT transitions and) net rewrites. A generalization of structural relations for rewritable PT is part of ongoing work.

Thus, a rewritable PT-net nicely extends the behaviour of the corresponding PT system: This directly comes from $\texttt{firing} \in \boldsymbol{R}$.

Property 2. $TS(\texttt{net m0}, \{\texttt{firing}\}) \subseteq TS(\texttt{net m0}, \boldsymbol{R})$

If, for modelling reasons, one wants to assign rules implementing system transformations a higher priority than `firing` rule she/he has two possibilities: either making `firing` match a `System` term only when no other rules do or using the `Maude`'s strategy meta-modules to set an order of evaluation for rules. We don't consider these topics here.

5. Formal verification of rewritable PT systems

This section briefly presents the tools available to formally verify models specified using rewritable PT nets and a few experimental data that refer to the two base components shown in Fig. 5. We will exploit some outcomes in the rewritable PT specification of a gracefully degrading production system.

5.1. Model checking

A `Maude` system module (which specifies a rewrite theory) provides an executable formal model for a distributed system. Under finite reachability assumption, we can model-check invariant properties (or get counterexamples) and, using ad hoc modules, efficiently verify linear time temporal logic (LTL) formulas. If the state-transition system associated with a ground term is very large or unlimited we may carry out bounded searches or use abstractions (typically, taking the form of extra state-equations).

Throughout the paper, we exploit a base, yet powerful, capability made available by the `Maude` run time support, namely, the formal verification of invariants through the `search` command. This tool explores the reachable state space of a term following a breadth-first strategy.

We conduct a preliminary analysis of the two PT nets in Fig. 1, separately taken. Each has simple formalization as rewritable PT with $\boldsymbol{R} = \{\texttt{firing}\}$. We are mainly interested in liveness properties. In particular, we aim to verify that both models eventually reach some final states (following a line's fault) and precisely characterize them.

A straightforward way to check for final states with `Maude` is issuing the command below, which searches for all *final* states reached by a `System` term:

```
search net m0 =>! X:System .
```

Or alternatively:

```
search net m0 =>* X:System such that dead(X:System) .
```

(symbol * means "in zero or more steps").

Table 2 summarizes the search results and performances, as the model's parameter (the number of raw pieces worked in a cycle) varies. All the data shown in this paper refer to an Intel Core i5 11th gen equipped with 40 GB of RAM. The first half of the table refers to the nominal configuration, the second half to the degraded one.

For both models and any initial configuration with $2 \cdot M$ pieces`search` has two solutions (i.e., final states). This prefigures a parametric (or structural) behaviour.

As for the nominal PL, the two final markings (`Pbag` sub-terms of `search` solutions) have a symmetric (i.e., permutable) form (we use the same labels/subscripts as in Fig. 1):

```
1 . p(< "f" ; 0 >) + M . p(< "w" ; 0 >) + M . p(< "a" ; 1 >)
1 . p(< "f" ; 1 >) + M . p(< "w" ; 1 >) + M . p(< "a" ; 0 > )
```

We can give an intuitive interpretation: A PL eventually enters a deadlock following a line's fault where half of the pieces are on the left line waiting for the assembly phase, and the other half (to be worked) is on the broken line.

As for the degraded PL (Fig. 1-bottom) – in which we assume that M raw pieces have moved from the faulty line–, the two final states look like these:

Table 2
`search` results for the models corresponding to Fig. 1.

$2 \cdot M$	states (final)	rewrites	time (ms)
2	15(2)	1329	0
4	42(2)	2850	0
6	90(2)	5744	4
8	165(2)	10457	8
10	273(2)	17435	12
Degraded configuration			
2	8(2)	705	0
4	18(2)	1088	0
6	32(2)	1657	0
8	50(2)	2412	2
10	72(2)	3353	3

Table 3
Semiflows of the PT nets in Fig. 1.

nominal		degraded	
P	$s+2*w_0+2*a_0$	P	$s+w_0+a_0$
P	$s+2*w_1+2*a_1$	P	$o+f_0$
P	$o+f_0+f_1$		
T	$ld+ln_0+ln_1+as$	T	$ld+2*ln_0+as$

```
1 . p(< "f" ; 0 >) + 2 * M . p(< "w" ; 0 >)
1 . p(< "f" ; 0 >) + (2 * M - 1) . p(< "w" ; 0 >) + 1 . p(< "a" ; 0 > )
```

In other words, the degraded PL eventually enters a deadlock following a second fault with the total of pieces to be worked except for at most one ready for assembly.

5.2. Exploiting PT net structure

In a hybrid formalism, we can benefit from the tools/techniques available for any employed formalism. Structural analysis of Petri nets may be a viable, efficient alternative/integration to state-space-based techniques when the system state-space is too large (unbounded) and the performances of analysis tools degrade. Another advantage of structural analysis is that it doesn't depend on the initial state of a system and can be used to infer parametric properties.

Typical structural techniques are the structural relations among PT nodes (e.g., conflict, causal connection, mutual exclusion) and structural invariants (semiflows). We apply semiflows to the two base components of our case study.

Let $\mathbf{Q}$ be the $|P|\cdot|T|$ matrix such that is $\mathbf{Q}_{[p,t]} = O(t)(p) - I(t)(p) \in \mathbb{Z}$. Any positive integer P-vector $\mathbf{p}$ which is a non-null solution of the product $\mathbf{p}\cdot\mathbf{Q} = \mathbf{0}$ is called P-semiflow and expresses a conservative law for the marking of places corresponding to non-zero entries of $\mathbf{p}$. Any positive integer T-vector $\mathbf{t}$ which is a non-null solution of the product $\mathbf{Q}\cdot\mathbf{t} = \mathbf{0}$, is called T-semiflow and expresses a null (cyclic) effect on markings by any firing sequence matching $\mathbf{t}$.

Table 3 shows the semiflows[7] of the PT nets in Fig. 1. The T-semiflows (last row, one for each model) represent the production cycle in the nominal and degraded PL. In the latter, the doubled activity of the working line is evident.

Because P-semiflows cover all places, the PT systems are *structurally* bounded. Using semiflows, we can confirm that the model-checking outcomes for the two components in Fig. 1 are valid for any value of M. Consider, e.g., the two P-semiflows of the degraded PL and its (parametric) initial marking (cf. Fig. 1, bottom): We derive these invariant expressions, for each reachable marking m:

$$\begin{aligned} m(o) + m(f_0) &= 1 \\ m(s) + m(w_0) + m(a_0) &= 2\cdot M \end{aligned}$$

With simple arguments, we figure out that if $m(f_0) = 1$ (meaning that a second fault has occurred) then the degraded PL eventually reaches (through a firing sequence of ld and as) either: $m' = 1 * f_0 + 2\cdot M * w_0$ or $m'' = 1 * f_0 + 1 * a_0 + (2\cdot M - 1) * w_0$, by the matches of the `search` command.

While it is not the focus of this paper, the opportunity to exploit Petri nets structural analysis in combination with the model-checking facilities available in `Maude` looks promising and deserves study, as discussed in Section 9.

6. Facing model scalability with advanced verification approaches: "symmetric" net operators

`Maude` rewritable PT nets are an expressive model for adaptive distributed systems. Nonetheless, we must integrate this formalism to face the complexity of large-scale models at the description and analysis/simulation levels. The net algebra we present in this section considers both aspects.

[7] Computed with the `GreatSPN` tool (github.com/greatspn/SOURCES).

Net (or Process) algebra operators are a classical approach to managing model complexity at a descriptive level using modularity/hierarchy. Rarely, however, modular specifications are matched by efficient analysis techniques, which limits their usability in practice. Almost always, a model built compositionally needs to be unfolded (flattened) to be analysed/simulated.

We introduce a few base net operators (a bit more general than process-oriented ones) that implicitly point out system symmetries by incrementally annotating net places with labels encoding the model's modular layout.

With simple, efficient manipulation of these annotations, it is possible to exploit the model's symmetries for building a compact *quotient reachability graph*. In the event of many replicated components, this approach outperforms in terms of space and time some recently presented, including ours fully integrated into `Maude` [13] and based on traditional graph canonization.

Net morphism Detecting behaviour-equivalent (sub-)systems boils down to graph *morphism*: A morphism between PT nets is a pair of bijections between their places and transitions preserving edges and transition labels.[8] A morphism between PT systems is a net morphism preserving the markings. Two PT nets/systems are said isomorphic ($\cong$) if a morphism maps one into the other. A morphism between nodes of the same net is called an *automorphism*.

When rephrasing these notions for `Net` and `System` terms (where we assume there are no isolated places), we have to take place labels into account:

```
op strLab : Place -> NeList{String} .
```

gets out the list of tags from a label. `places` is an overloaded operator resulting in the set of places in a `Pbag` or a `Net`.

Definition 2 *(`Net` (`System`) morphism, permutable sets).* Let `N`, `N'` and `m`, `m'` be `Pbag` and `Net` ground terms, respectively:

A *morphism* between `N` and `N'` is a bijection ϕ : `places(N)` $\rightarrow$ `places(N')` such that ϕ(`N`) = ϕ(`N'`)[9] and `strLab`(ϕ(`p`)) = `strLab`(`p`) for each `p`. If `N = N'` then ϕ is said an *automorphism*. A morphism between `N m` and `N' m'` is a morphism ϕ between `N` and `N'` such that ϕ(`m`) = `m'`.

Let $S, S' \subseteq$ `places(N)`. S and S' are *automorphic* if and only if there exists an automorphism ϕ such that the image under ϕ of S is S'.

S and S' are *permutable* if and only if they are automorphic, and the partial mapping $S \rightarrow S'$ extended as an identity to the other places is an automorphism. If $S = S'$, then S is said self-permutable.

Clearly, if S and S' are permutable, then $S \cup S'$ is self-permutable.

6.1. Structured labelling and modular symmetries

A library of net operators allows us to build any `System` (`Net`) in a modular way. These operators encode the nesting of similar components through place annotations defined incrementally. Place labels highlight the hierarchy of component replicas and implicitly the model's symmetry structure. By convention, the root of the hierarchy corresponds to the label's suffix.

Notation In the rest of the section, `N` denotes a ground term of sort `Net`, `m` a `Pbag`, `L` a (possibly empty) `Lab`, `NeL` a non-empty label (a term of sort `NeLab < Lab`), `w` a `String` and `i`, `j` two `Nat`. We admit variants with superscripts and use non-final terms as patterns: for example, `L < w ; i >` matches any label ending with `< w ; i >`, including `< w ; i >`.

Let S be a set of `Place` terms, namely, a `Pset` term: S_{NeL} denotes the subset of S whose labels have the *suffix* `NeL`.

Later, we may use as subscript of S a term of sort `NeList{String}`: $S_{\text{NeLw}} := \bigcup_{\text{NeL}:\ \text{strLab(NeL)==NeLw}} S_{\text{NeL}}$, $S_{\{\text{NeLw}\}} :=$ {`p(NeL)` | `strLab(NeL) == NeLw`}.

Definition 3 *(Symmetric Labelling).* A `Net N` has a symmetric labelling if and only if any two sets `places(N)`$_{\text{<w ; i> L}}$, `places(N)`$_{\text{<w ; j> L}}$ are permutable. A `System (N m)` has a symmetric labelling if and only if `N` has.

In a symmetrically labelled `Net`, maximal sets of places whose labels have the same suffix (possibly empty), preceded by elementary labels with the same tag, are permutable.

Definition 3 provides an intuitive characterization of symmetries reflecting the model's modular definition. We can describe it using a tree of `Strings`, where each level corresponds to a nesting level in place labels (and in the model). Sub-trees with similar roots coming from the same branch represent permutable sets. Descending through a tree corresponds to "navigating" place labels right-to-left.

Definition 3 induces increasingly refined partitions of net places into permutable classes and has nice corollaries, e.g., letting `N` meet Definition 3:

- For any `NeL`, the set `places(N)`$_{\text{NeL}}$ is self-permutable.

---

[8] The extension of $\phi : A \rightarrow A$ induced on multisets is: let $b \in Bag[A]$, $\phi(b)(a) = b(a)$ for each $a \in A$.

[9] The extension of ϕ induced on `Nets` must preserve `Transition` labels.

- For any two `NeL`, `NeL'` such that `strLab(NeL) == strLab(NeL')`, sets $\texttt{places(N)}_{\text{NeL}}$ and $\texttt{places(N)}_{\text{NeL}'}$ are automorphic.
- Any two places `p(< w ; i > L)` and `p(< w ; j > L)` are permutable.

A concrete example of a symmetry tree is provided at the end of Section 6.3.

We are not interested in the algebraic characterization of symmetries as (sub)groups. Instead, we formalize a few intuitive, operationally worthy lemmas.

From now on, when speaking of net morphism we will implicitly refer to `Net` or `System` ground terms that meet Definition 3.

(Bulk) contextual swap If the `Net` underlying a `System` meets Definition 3, we can truly efficiently bring `System` terms to a *normal* form through a *contextual* index-swap on place labels. The arity of the base version is:

```
op swap : Pbag String Lab Nat Nat -> Pbag .
```

`swap(m,w,L,i,j)` is the obtained from `m` by swapping `< w ; i >` $\leftrightarrow$ `< w ; j >` in $\texttt{places(m)}_{<w\,;\,i>\,L}$ and $\texttt{places(m)}_{<w\,;\,j>\,L}$: This operation affects the head of label suffixes. Clearly, if $\texttt{places(m)}_{<w\,;\,i>\,L}$ or $\texttt{places(m)}_{<w\,;\,j>\,L}$ are empty then `swap` does a contextual index-replacement.

`swap(N,w,L,i,j)` acts on $\texttt{places(N)}_{<w\,;\,j>\,L}$ and $\texttt{places(N)}_{<w\,;\,j>\,L}$ similarly. `swap(N m,w,L,i,j)` is defined as `swap(N,w,L,i,j) swap(m,w,L,i,j)`.

Bulk contextual swap has a few intuitive properties, summarized as follows (we assume that in a term `N m` : `places(N)` $\supseteq$ `places(m)`):

Corollary 1. *Let the* `Net` *term* `N` *meet Definition 3. Then:*
`swap(N,w,L,i,j)` $=$ `N'` $\cong$ `N` *and* `N'` *meets Definition 3.*
`swap(N m,w,L,i,j)` $=$ `N' m'` $\cong$ `N m` *and* `N'` *meets Definition 3.*

It directly follows from Definition 3: $\texttt{places(N)}_{<w\,;\,i>\,L}$ and $\texttt{places(N)}_{<w\,;\,j>\,L}$ (when non-empty) are permutable, likewise the outcomes of contextual index swap (the case in which one of these sets is empty is trivial). Clearly, `swap` preserves symmetric labelling. As a consequence, if we coherently `swap` a `Net` and its marking we get an isomorphic `System`. □

Name abstraction is a base technique in graph canonization. In this context, it boils down to index abstraction.

Definition 4 *(I-abstraction).* Let `t` be a ground term of sort `Pbag` or `Net`. We say that `t` has an I-abstract form if and only if:
`p(L < w ; i > L')` $\in$ `places(t)` $\Rightarrow \forall$ `j` $<$ `i` `p(L < w ; j > L')` $\in$ `places(t)`.

In other words, in an I-abstract `Net` or `Pbag` term label indices take consecutive values starting from zero, at each nesting level.

Note that the marking `m` of an I-abstract net `N` may not be I-abstract. For example, the marking `2 . p(< "w" ; 1 >)` of the `Net` described in Listing 2 is trivially non-I-abstract.

Corollary 2. *Let* `N` *meet Definitions 3, 4.*
If neither $\texttt{places(N)}_{<w\,;\,i>\,L}$ *nor* $\texttt{places(N)}_{<w\,;\,j>\,L}$ *is empty then:*
`swap(N,w,L,i,j)` $=$ `N`.
`N swap(m,w,L,i,j)` $\cong$ `N m`.

`swap` acts on sets of permutable places mapping one into the other by preserving adjacency and the text component of labels (definition of morphism). Due to index abstraction, `swap` *exactly* maps one set into the other. The 2nd point comes from: `N swap(m,w,L,i,j)` $=$ `swap(N m,w,L,i,j)` and Corollary 1. □

A symmetrically labelled `Net` has a unique I-abstract form.

Corollary 3. *Let* `N` *meet Definition 3. Then, there is a unique* `N'` $\cong$ `N` *that meets Definitions 3, 4.*

Let `p(L' < w ; i > L)` $\in$ `places(t)` and exists `j` $<$ `i` : `p(L' < w ; j > L)` $\notin$ `places(t)`: According to Definition 3, the set $\texttt{places(N)}_{<w\,;\,j>\,L}$ is empty. To preserve Definition 3 we have to bulk-replace `i` with `j` in $\texttt{places(N)}_{<w\,;\,i>\,L}$. This operation is a particular contextual swap: By repeatedly applying it we eventually get out an I-abstract form which preserves Definition 3 (see the above corollaries). Independently on which `j < i` we start from, we converge to the same form, which is unique, according to Corollary 2. □

The above properties have practical relevance: If we know, e.g., that the `Net` sub-term of a `System` has symmetrical labelling and I-abstract form, then we only have to normalize its marking. See also Section 6.4.

Tools for net composition (Compositional) Net operators directly or indirectly build on the associative `Net` *juxtaposition*: _ ; _ which implicitly *folds* places shared between operands. Another base operation in the net composition is *merging* (*gluing*) nodes of a net.[10] These operations general do not preserve Definition 3: In the sequel, we set a few simple conditions for retaining symmetric labelling that net operators implicitly use. Indeed, preserving symmetric labelling by net operators is the key to our approach.

Property 3. *Given a* `Net N` *meeting Definition 3:*

1. *For any* `w`*, the* `Net` *obtained by merging* $\texttt{places(N)}_{\{w\}}$ *(if non-empty) into a place* `p(< w ; 0 >)` *meets Definition 3.*
2. *For any* `NeLw`*, the* `Net` *obtained by merging* each *set* $\texttt{places(N)}_{\text{NeL}}$ *such that* `strLab(NeL) == NeLw` *into* `p(NeL)` *meets Definition 3.*
3. *For any transition label* `X`*, the* `Net` *obtained by merging the* `Transitions` *with label* `X` *meets Definition 3.*

The first point comes from $\texttt{places(N)}_{\{w\}}$ (holding places `p(< w ; i >)`) being self-permutable (and Definition 3). As for the second point, any non-empty list of strings `NeLw` induces a partition of $\texttt{places(N)}_{\text{NeLw}}$ into (self-)permutable classes: Each class holds places with a label suffix `NeL` matching `NeLw` and can be coherently merged into `p(NeL)`. The third point directly follows from `Net` morphism having to preserve transition labels. □

In most practical cases, we have to merge places with elementary labels. The following definition sets a simple condition under which `Net` juxtaposition retains symmetric labelling.

Definition 5 *(Suffix-disjoint form).* Two nets `N, N'` are in *disjoint form* if and only if for any two places `p(L < w ; i >)` ∈ `places(N)`, `p(L' < w' ; j >)` ∈ `places(N')` : `w =/= w'`.

The `Maude` formalization of the definition above as a predicate is:

Listing 4. Predicate testing Definition 5.

```
op sdisjoint : Net Net -> Bool [comm] .
ceq sdisjoint(N, N') = false if p(L < w ; i > ) U S := places(N) /\
   p(L' < w ; j > ) U S' := places(N') /\ L < w ; i > =/= L' < w ; j > .
 eq sdisjoint(N, N') = true [owise] .
```

Property 4. *If* `N, N'` *meet Definitions 3, 5. then* `N ; N'` *meets Definition 3.*

The condition on the suffix of *different* places of `N` and `N'` makes their labels meet Definition 3 also after juxtaposition (shared places trivially do as well). □

Intuitively, `Net` juxtaposition retains symmetric labelling if any pair of different places of the nets to join have label tails with different tags. This condition is quite restrictive, as well as sharing single places.

A generalization consists of merging *classes* of (permutable) places through a sort of *Cartesian product*. This helps in net-algebra compositions. To avoid cumbersome relabelling, we limit to places with elementary labels. This usually fits with modelling needs (we might set more general conditions).

Property 5. *Let* `N` *meet Definition 3 and* `w =/= w'`*. If* $A := \texttt{places(N)}_{\{w\}}$ *and* $B := \texttt{places(N)}_{\{w'\}}$ *are non-empty then the* `Net` *obtained by merging each pair* $p \in A$*,* $p' \in B$ *so that the resulting places carry elementary labels with tag* `w''` *such that* ∄ `p(L < i ; w'' >)` ∈ `places(N)` $- A - B$*, meets Definition 3.*

The elements of A and B are permutable: The outcomes of merging each pair `p` $\in A$, `p'` $\in B$, inherit connections, therefore, are permutable in turn and (according to the hypothesis) labelled coherently with Definition 3, likewise the others. □

Here is an intuitive corollary of Property 5.

Corollary 4. *Let* `N, N'` *meet Definition 3 and* `w =/= w'`*. If* $A := \texttt{places(N)}_{\{w\}}$*,* $B := \texttt{places(N')}_{\{w'\}}$ *are non-empty and* `N`$-A$*,* `N'`$-B$ *meet Definition 5 then the* `Net` *obtained from* `N ; N'` *by merging* A *and* B *meets Definition 3.*

[10] Merging (gluing) graph nodes corresponds to replacing them with one which inherits their adjacency relations. In a multiset-based encoding of nets, merging places means identifying them on every bag with a fresh new one whose multiplicity is the sum of their multiplicities. Merging `Transitions` means replacing two or more such terms with one whose `Tmatrix` sub-term is the component-wise sum of their matrices.

Listing 6 summarizes the main *join* and *merge* operators, some of which exploit the properties above. `join` is overloaded to allow for `System` juxtaposition. Note that `sjoin` (which preserves symmetrical labelling) is defined as a *partial function* using a *kind* for its range.

Listing 5. Base tools for Net composition.

```
*** Parameter X represents the type of transition labels ***

*** merge operators (based on Properties 3, 5)
op merge : Net Set{X} -> Net . *** merge specific identically labelled transitions
op merge! : Net -> Net . *** merge all identically labelled transitions
op merge : Net NeList{String} -> Net . *** merges classes of permutable places
op merge : Net String -> Net . *** merge places with similar elementary labels
...
*** join operators
op join : System System -> System . *** builds on ';', sums up markings (I)
op join : System System Set{X} -> System. *** joins by merging some transitions
op join! : System System -> System . *** builds on merge!
op sjoin : Net Net -> [Net] . *** Symmetry preserving juxtaposition (II)
op sjoin : Net Net String String String -> [Net] . *** Corollary 4
op sjoin : System System -> [System] . *** system version
...
*** Examples of definitions
vars N N' : Net .
vars B B' : Pbag .
eq join(N B, N' B') = (N ; N') B + B' . *** (I)
ceq sjoin(N, N') = N ; N' if sdisjoint(N, N') . *** (II)
```

6.2. (Compositional) net-operators

Below is a commented list of the leading net-algebra operations. Those with prefix `"repl"` have particular relevance: They replicate a given `System/Net` component coherently with Definition 3. They work by incrementally appending new elementary labels to place labels. Besides a base version, two others mime the parallel (fork/join) and alternative replication. Listing 6 includes introspection/intercession operators employed in rewrite rules and classic (binary) process algebra operators, defined as *partial* functions (as `sjoin`, upon which they rely).

Looking at operator arity, we refer to places through a pattern –a (non-empty list of) `String`– that label suffixes have to match, coherently with the preservation of the symmetry structure (Definition 3). For example, `"L" "PL"` represents all places of sub-component `"L"` in the hierarchy rooted by `"PL"`.

For space reasons, Listing 6 doesn't include overloaded operators, e.g.:

- Some operators with range or arity `System` (e.g., replicators) have `Net` counterparts, defined as `eq netop(N,..) = net(sysop(N nilP,..))`. Vice-versa, operators with range or arity `Net` (e.g., `merge`) have `System` counterparts with intuitive definition.
- Bulk versions with `Set{T}` in place of type `T` in the arity are available.

Proposition 1. *If the* `Net` *(or* `System`*) operands of a ground term having as top operator one in Listing 6 meet Definition 3 then also the term's final form does, if of sort* `Net` *(or* `System`*).*

The proof is merely technical and relies on the (likely inductive) definition of operators. Let us outline it for the core operator `repl&share`, whose definition is given at the end of Listing 5. The operator's evaluation is the default one: Equations first apply to operands and then to the top operator. We may thus assume that `Sys` operand is a canonical `System` that meets Definition 3. The operator's tail-recursive definition builds on `$repl&share`, which repeatedly `join`s to the partial result (`Sys'`, initially empty) a term built from `Sys` by appending (`addLab`) an element `< W ; I >` to all place labels except for those with suffix tags matching `LW`: This operation preserves symmetric labelling (once verified that the suffix tail's tag is other than `W`). Therefore, we end up showing that `join` preserves it as well: This is evident since we juxtapose `K` replicas (of a symmetrically labelled `System`) which differ in the index of the tail of place labels (ranging from `K - 1` down to 0). Transitions with given labels (`Z`) may be merged, but this operation retains the symmetry structure (Property 3). During juxtaposition, we repeatedly and exhaustively fold pairs of shared places carrying elementary labels: this operation, again, preserves symmetric labelling. □

Therefore, any `Net` (or `System`) terms we build/manipulate using the operators described in Listing 6 meet Definition 3 provided that we start from elementary nets (i.e., `Transition` terms) that verify it.[11]

[11] A constructor of elementary nets is available to ensure this.

Listing 6. A library of net operators.

```
              *** "Symmetric" replicators ***
*** Create K disjoint replica of a System by appending to place labels in each replica a tag (with index 0 through K−1) denoting the nesting level.
    Places (with elementary labels) and transitions to share may be specified (last two parameters).
op repl&share : System NzNat String String Set{X} −> System .
op repl : System NzNat String −> System . *** Default: no places to share.
op replAnd : System NzNat String String X$Elt String X$Elt String String −> System .
op replOr : System NzNat String String String −> System .

*** Elementary builder: Creates a transition that meets Def.2,3 ***
op newT : String Nat String Nat String Nat X$Elt −> Transition .

            *** Classic Process−Algebra ops ***
*** PAR composition: 3rd and 4th args: in/out places (they have to match the same label in both nets), 5th, 6th: tags of the fork transition and its
    input place, respectively. 7th, 8th: The same for the join transition and its output place
op and : System System String String X$Elt String X$Elt String −> [System] .
*** ALT composition (see the description of "and")
op or : System System String String −> [System] .
*** SEQ composition
op seq : System System String String −> [System] .

           *** "Symmetric" introspection/intercession ops ***
*** Gets net places whose labels match a given suffix
op get : Net NeList{String} −> Pset .
*** Gets the sub−net built of transitions with labels in the specified set
op subNet : Net Set{X} −> Net .
*** Sets up a marking for net places whose labels match a given suffix
op setMark : System NeList{String} Nat −> System .
*** Associates a marking to a net
op setMark : Net NeList{String} Nat −> System .
*** Sets up edges of specific type/weight (0 weight means removing)
op set : Net X$Elt Atype NeList{String} Nat −> Net .
*** Sets up edges of specific type/weight linking a transition to a set of k newly added places with elementary labels and indices 0 through k−1.
    Adds the transition if it doesn't exist yet. Returns the same Net if the induced action would violate Def. 2
op set : Net X$Elt Atype String NzNat NzNat −> Net .
*** Removes net places whose labels match a given suffix
op remove : Net NeList{String} −> Net .
*** Removes net's transitions with a specific label
op remove : Net X$Elt −> Net .
*** Joins a new component (sub−net, 2nd arg) to the specified net (1st arg) at the highest level of hierarchy (root), represented by a suffix tag: if
    isomorphic components do exist, then the prefix header is associated with a suitable index. Returns the same Net if the induced action
    would violate Def. 2
op attach : Net Net String −> Net .
op detache : Net Net String −> Net . *** Does the opposite of attach

*** Example of definition: repl&share
vars Sys Sys' : System .
var K : NzNat .
vars W W' : String .
var S : Pset .
var Z : Set{X} .
eq repl&share(Sys,K,W,W',Z) = $repl&share(Sys,K,W,emptySys, places(Sys, W'), Z) .
op $repl&share : System Nat String System Pset Set{X} −> System .
eq $repl&share(Sys, 0, W, Sys', S, Z) = Sys' .
eq $repl&share(Sys, K, W, Sys', S, Z) = $repl&share(Sys, K − 1, W, join(Sys', addLab(Sys, < W ; K − 1 >, S), Z), S, Z) .
```

6.3. Example: fault-tolerant distributed production system

We illustrate how compositional net operators work through the formal specification of a distributed (gracefully degrading) Production System.

Nominally, the system consists of N Production Units (PL, see Fig. 1) sharing a warehouse of raw pieces. Each PL is composed in turn of K interchangeable production lines, or robots (in our case study, $K = 2$). The system's graceful degradation is specified in Section 6.5.

Listing 7. Modular specification of a fault tolerant production system.

```
fmod PT−FTPL is
 protecting NET−OP{String} * (op emptyStlab to emptyStr) .
 ops line load ass fault faultyload faultyass : −> Transition .
 ops cycle faultycycle : −> Net .
 ops PL PLA nomPL faultyPL NfaultyPL : NzNat −> Net . *** PL with K lines
 op faultySys : NzNat −> System .
 op NPL : NzNat NzNat −> Net . *** N PLs (0 .. N− 1) each with K lines (0 .. K−1)
 op NPLsys : NzNat NzNat NzNat −> System . *** N PLs with K lines and K * M tokens in warehouse place
 ops NfPL : NzNat −> Net . *** N degraded PLs
 vars N M K : NzNat .
 eq load = [1 . p(< "s" ; 0 >) , 1 . p(< "w" ; 0 >) , nilP ] |−> "ld" .
 eq line = [1 . p(< "w" ; 0 >) , 1 . p(< "a" ; 0 >), 1 . p(< "f" ; 0 >) ] |−> "ln" .
 eq ass = [1 . p(< "a" ; 0 >) , 1 . p(< "s" ; 0 >) , nilP ] |−> "as" .
 eq fault = [1 . p(< "o" ; 0 >) , 1 . p(< "f" ; 0 >), nilP ] |−> "ft" .
 eq cycle = load ; line ; ass .
 eq PL(K) = repl&share(cycle ; fault, K, "L", p (< "o" ; 0 >) U p(< "s" ; 0 >), "as" U "ld") . *** PL built of K symmetric lines
 eq NPL(N, K) = repl&share(PL(K), N, "PL", p(< "s" ; 0 >), emptyStr) .
 eq NPLsys(N, K, M) = setMark(setMark(NPL(N, K), "o" "PL", 1), "s", K * M) .
 ...
endfm
```

We start with elementary nets (`Transition` terms) meeting Definitions 3, 4[12] and use aliasing for readability. We first specify a production unit's control flow (the production cycle) using the first time `repl&share` operator (which builds on `join`). The parametric alias `PL(K)` denotes a PL composed of `K` similar lines: if we `reduce PL(2)` we get a `Net` identical to the PT in Fig. 1 (top) but for richer labelling: The (sub)model's hierarchy is expressed by automatically appending an elementary label with tag `"L"` to place labels, e.g., `p(< "a" ; 0 > < "L" ; 1 >)` is the "assembly" place of line 1 of the PL. We may optionally exclude some places that are shared among replicas: In this case, those representing the "warehouse" of pieces and fault occurrence. We also indicate transitions to share among replicas: "load" and "assembly". As an effect of `join`, the input/output edges from the shared "warehouse" are rightly weight-two (K). Below is the output of `reduce` command, which brings a term into canonical form.

```
Maude> reduce in FT−PL : PL(2) .
result Net:
[1 . p(< "o" ; 0 >),1 . p(< "f" ; 0 > < "L" ; 0 >),nilP] |−> "ft" ;
[1 . p(< "o" ; 0 >),1 . p(< "f" ; 0 > < "L" ; 1 >),nilP] |−> "ft" ;
[1 . p(< "w" ; 0 > < "L" ; 0 >), 1 . p(< "a" ; 0 > < "L" ; 0 >),1 . p(< "f" ; 0 > < "L" ; 0 >)] |−> "ln" ;
[1 . p(< "w" ; 0 > < "L" ; 1 >),1 . p(< "a" ; 0 > < "L" ; 1 >),1 . p(< "f" ; 0 > < "L" ; 1 >)] |−> "ln" ;
[2 . p(< "s" ; 0 >),1 . p(< "w" ; 0 > < "L" ; 0 >) + 1 . p(< "w" ; 0 > < "L" ; 1 >),nilP] |−> "ld" ;
[1 . p(< "a" ; 0 > < "L" ; 0 >) + 1 . p(< "a" ; 0 > < "L" ; 1 >),2 . p(< "s" ; 0 >),nilP] |−> "as"
```

The alias `NPL(N, K)` denotes a PT net composed of N PLs each with K lines. We get it by applying the second time the operator `repl&share` to the sub-net just described: The further nesting level is expressed by the tag `"PL"` appended to place labels, e.g., `p(< "a" ; 0 > < "L" ; 0 > < "PL" ; 1 >)` is the "assembly" place at line 0 of PL 1. Note, once again, the use of the operator's sharing option to make PLs pick up raw pieces, two (K) at a time, from the common warehouse. `NPL(2,2)`, e.g., corresponds to the PT net at the top of Fig. 4. Finally `NPLsys(N, K, M)` is the PT system obtained by putting $K \cdot M$ tokens in the "warehouse" place and one token in the places modelling fault occurrence on each PL (those with tag `"o"`). The `reduce` outcome of e.g. `NPLsys(2,2,2)` is given in Fig. 2.

Using symmetric PA operators We may use the symmetric version of process algebra operators to get the same model. For example:

Listing 8. Alternative specification using process algebra.

```
eq PLA(K) = merge(replAnd(line ; fault,K,"w","a","ld","s","as", "s", "L"), "o" "L") .
eq PL(K) = set(set(PLA(K), "ld", i, "s", K), "as", o, "s", K) .
eq NPL(N, K) = replOr(PLA(K), N, "s", "s", "PL") .
...
```

The final net is the combination of a symmetric "And" (for the K production lines of a PL and their faults) and an "Or" (for the N PLs). The "And" implicitly adds a pair of fork/join transitions corresponding to "load" and "assembly", respectively, along with their input/output places, which coincide with the warehouse: This is why `"s"` (which labels that place) occurs twice in both the "And" and "Or". The "And" is doubly wrapped with `merge` –which folds the places triggering faults (tag `"o"`) into a single one– and `set`

[12] We could get them using ad-hoc operators like `newT` or `set`.

```
Maude> reduce in FT-PL : NPLsys(2, 2, 2) .
result System: ([1 . p(< "o" ; 0 > < "PL" ; 0 >),1 . p(< "f" ; 0 > < "L" ; 0 > < "PL" ; 0 >),nilP]
|-> "ft" ; [1 . p(< "o" ; 0 > < "PL" ; 0 >),1 . p(< "f" ; 0 > < "L" ; 1 > < "PL" ; 0 >),nilP]
|-> "ft" ; [1 . p(< "o" ; 0 > < "PL" ; 1 >),1 . p(< "f" ; 0 > < "L" ; 0 > < "PL" ; 1 >),nilP]
|-> "ft" ; [1 . p(< "o" ; 0 > < "PL" ; 1 >),1 . p(< "f" ; 0 > < "L" ; 1 > < "PL" ; 1 >),nilP]
|-> "ft" ; [1 . p(< "w" ; 0 > < "L" ; 0 > < "PL" ; 0 >),1 . p(< "a" ; 0 > < "L" ; 0 > < "PL" ; 0 >),1 . p(< "f" ; 0 > < "
    L" ; 0 > < "PL" ; 0 >)]
|-> "ln" ; [1 . p(< "w" ; 0 > < "L" ; 0 > < "PL" ; 1 >),1 . p(< "a" ; 0 > < "L" ; 0 > < "PL" ; 1 >),1 . p(< "f" ; 0 > < "
    L" ; 0 > < "PL" ; 1 >)]
|-> "ln" ; [1 . p(< "w" ; 0 > < "L" ; 1 > < "PL" ; 0 >),1 . p(< "a" ; 0 > < "L" ; 1 > < "PL" ; 0 >),1 . p(< "f" ; 0 > < "
    L" ; 1 > < "PL" ; 0 >)]
|-> "ln" ; [1 . p(< "w" ; 0 > < "L" ; 1 > < "PL" ; 1 >),1 . p(< "a" ; 0 > < "L" ; 1 > < "PL" ; 1 >),1 . p(< "f" ; 0 > < "
    L" ; 1 > < "PL" ; 1 >)]
|-> "ln" ; [2 . p(< "s" ; 0 >),1 . p(< "w" ; 0 > < "L" ; 0 > < "PL" ; 0 >) + 1 . p(< "w" ; 0 > < "L" ; 1 > < "PL" ; 0 >),
     nilP]
|-> "ld" ; [2 . p(< "s" ; 0 >),1 . p(< "w" ; 0 > < "L" ; 0 > < "PL" ; 1 >) + 1 . p(< "w" ; 0 > < "L" ; 1 > < "PL" ; 1 >),
    nilP]
|-> "w" ; [1 . p(< "a" ; 0 > < "L" ; 0 > < "PL" ; 0 >) + 1 . p(< "a" ; 0 > < "L" ; 1 > < "PL" ; 0 >),2 . p(< "s" ; 0 >),
    nilP]
|-> "as" ; [1 . p(< "a" ; 0 > < "L" ; 0 > < "PL" ; 1 >) + 1 . p(< "a" ; 0 > < "L" ; 1 > < "PL" ; 1 >),2 . p(< "s" ; 0 >),
    nilP]
|-> "as") 1 . p(< "o" ; 0 > < "PL" ; 0 >) + 1 . p(< "o" ; 0 > < "PL" ; 1 >) + 4 . p(< "s" ; 0 >)
```

Fig. 2. `reduce` outcome of `NPLsys(2,2,2)`.

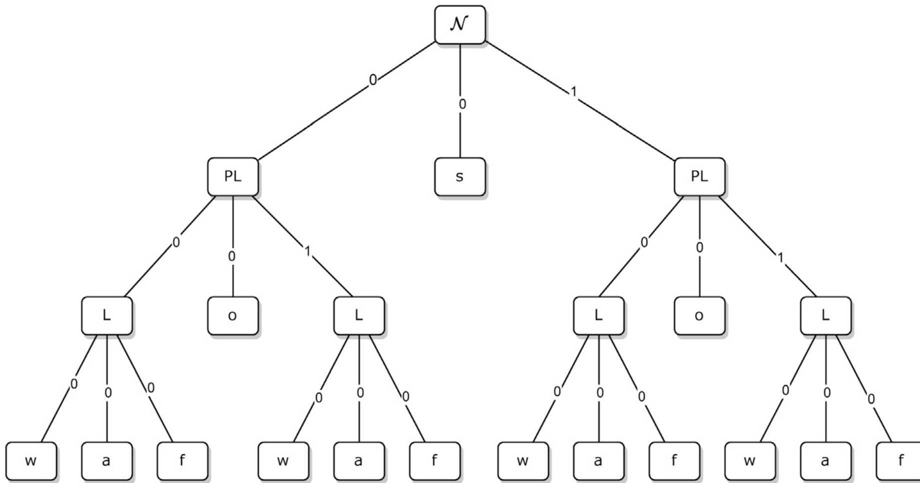


Fig. 3. The symmetry hierarchy of `NPL(2,2)`.

–making the I/O edges from/to warehouse are weight-two (K). The "Or" makes the PL replicas share the input place(s) with label `"s"`.

The model's symmetry structure illustrated Fig. 3 shows the symmetry structure of `NPL(2,2)` (a net composed of two PLs each holding two lines). The special symbol $\mathcal{N}$ marks the tree's root. The strings used in place labels mark the other nodes. Edges carry index values which refer to the target nodes (in the example, index values match Definition 4). At each level, sub-trees with similar roots and departing from the same branch represent permutable sets. The leaves of the tree correspond to the places.

6.4. Normalization procedure

We aim to build a quotient reachability graph by bringing `System` terms built and managed using symmetry preserving operators to a normalized form $\hat{s}$, such that, if s_1, s_2 meet Definition 3, $s_1 \cong s_2 \Leftrightarrow \hat{s}_1 = \hat{s}_2$, analogously to what we have done in [13] but incomparably more efficiently. Conversely, the canonization technique [13] is general, i.e., doesn't take labelling into consideration. Also from the modelling perspective, however, node labelling is highly beneficial and makes the theoretical loss of generality irrelevant.

The normalization procedure is implemented by a few operators which rely upon Definitions 3, 4 and related properties, that we summarize as follows:

Corollary 5. *Let the* `Net` *term* `N` *meet Definitions 3, 4. Then* `N` *is the* most minor *term in the class of isomorphic ones meeting Definition 3, according to any total order based on the bulk contextual index* `swap` *(Section 6.1).*

The reason why the I-abstract form of a symmetrically labelled Net coincides with the minimal one, according to any total order based on swap, is clearly because it represents the fixed point of this operation.

Here is the list of normalization operators, the leading of which build on (bulk) contextual swap:

Listing 9. Normalization operators (with some definitions).

```
*** Normalizes a marking: minimization plus index abstraction (I)
op normalize : Pbag -> Pbag .
*** Brings a marking (of a symmetrically labelled net) to a minimal form (II)
op minimize : Pbag -> Pbag [memo] .
op abstract : Pbag -> Pbag . *** Marking index abstraction
op abstract : System -> System . *** System index abstraction
*** Normalizes a System assuming that the Net sub-term is symmetrically labelled: applies only index abstraction to the latter (III)
op normalize : System -> System .

*** Gets the sub-bag of elements with given suffix and suffix header's indices
op subag : Pbag String Set{Nat} Lab -> Pbag .
*** Strict total order defined on bags.
op le : Pbag Pbag -> Bool .

*** Examples of definition (variables' sorts may be inferred by operator arity)
eq normalize(B) = minimize(abstract(B)) . *** (I)
ceq minimize(K . P + K' . P' + B) = minimize(B' + (B - B"))
 if p(L' < W ; I > L) := P /\ p(L" < W ; J > L) := P' /\ J > I /\ (K =/= K'
 or-else L' =/= L") /\ B" := subag(B, W, I U J, L) /\ B' := swap(B" + K . P + K' . P', W, L, I, J) /\ le(B', B" + K . P + K' . P') . *** (II)
eq minimize(B) = B [owise] . *** (II)
ceq normalize(Sys) = net(Sys') normalize(marking(Sys')) if Sys' := abstract(Sys). *** (III)
```

Index-abstraction and normalization operators are overloaded: One version works on markings (Pbag sub-terms), the other on whole System terms. Indeed, we only need to normalize the marking of a System following an occurrence of firing rule. Marking normalization is the core operation, including index abstraction and *minimization*[13]: The latter brings a Pbag to a minimal form, according to the total order implemented by operator le (based on swap and a total order among Place terms), which straightforwardly induces total orders on Net and System terms. The minimize operator assumes that the Net operand is symmetrically labelled. It has shown very efficient despite its compact and intuitive definition (included for reader convenience) which builds on the contextual swap on labels and le.

Compared to graph canonization, choices carried out at each rewriting step are "deterministic": No pruning strategy and no backtrack is required.

A System term normalization (following any rewrite rule changing the underlying Net) includes index abstraction and normalization circumscribed to Pbag sub-term. This simple technique exploits the preservation of symmetrical labelling by net operators and Corollary 5 and leaves room to further optimization.

Symmetry preserving net rewrites We aim to *efficiently* build a quotient graph associated with a rewritable PT net specified using compositional, or more generally, symmetry-preserving operators. The nodes of this graph are normalized System terms.

Normalization allows for the automatic detection of behavioural equivalences of a PT system. Equivalences are of two types: those due to the inner dynamics of a PT system (expressed by the firing rule) and those due to the structural changes that a PT system undergoes (specified by other rewrite rules).

In a dynamic context, rewrites rules *need* to preserve symmetric labelling if we want to exploit System terms normalization during state space searches.

We restrict ourselves to top rules, i.e., rewrite rules of System or Net type.

Definition 6 *(Symmetry preserving rule).* A top rule r is symmetry preserving if and only if for any ground term t meeting Definition 3, if $\mathtt{t} \overset{\mathtt{r}}{\Rightarrow} \mathtt{t}'$ then $\mathtt{t}'$ meets Definition 3.

The firing rule trivially meets the definition above. Let's focus on rules modifying the Net sub-term of a System.

There are many different ways of writing rules that meet Definition 6: We give a quite general syntactical characterization for System rules (straightforwardly adaptable to Net type rules).

Definition 7 *(Well-defined rule).* A rewrite rule $r : \mathtt{s} \Rightarrow \mathtt{s}'$ if $< cond >$ is well-defined (W.D.) if and only if $\mathtt{s}'$ is, according to the inductive definition:

[13] We have separated these steps to keep coding as simple as possible.

- s, the `Net` sub-term of a W.D. `System` term and a `System` whose `Net` is W.D. are W.D.
- Any term whose top operator is a symmetry-preserving one (Listing 5) and whose arguments of `System` and `Net` sort are W.D. is W.D.
- a `System` or `Net` free variable which is bound in a matching equation to a W.D. term is W.D.

It's quite natural to follow this pattern for rewrite rules. A non-trivial example is provided in Section 7.2.

Proposition 2. *Let r be a well-defined rule. Then r meets Definition 6.*

Let σ be a ground substitution of r's variables such that $\sigma(\mathrm{s})$ ($\mathrm{s} \in T_{\Sigma(X),\mathrm{System}}$) meets Definition 3. Then, according to Definition 7, also $\sigma(\mathrm{s}')$ does.

Since a quotient graph has to retain the base reachability properties of the original state-transition system, we need to integrate Definition 6. As shown in [13], it essentially boils down to using parametric rules built (only) of variable terms and symmetric operators (like inequalities). In our context, we may relax a bit on this requirement:

Definition 8 *(Parametric rule)*. A rule `r` of `System` type is *parametric* if and only if the ground terms that may occur on `r` are uniquely those of sort `List{String}`, `X$Elt` (for transition labels), `Atype` (denoting the PT arc types) used as arguments for symmetry-preserving operators, and the only operators involving `Nat` sub-terms of `Place` terms are `==`, `=/=`.

It stems from the observation that any `Net` or `System` term is made up of sub-terms of sort `Place`, which are in turn in one-to-one correspondence with `NeLab` terms (i.e., non-empty lists of pairs `Nat`, `String`).

Normalization procedure Let us sketch the normalization technique for RwPT:

1. We start rewriting (`search`ing) from a normalized `System` term.
2. We wrap the `Pbag` sub-term of `firing` rule's right-hand side with `normalize`:

```
N M => N normalize(M + O - I) if [I,O,H] |-> L ; N' := N
/\ I =/= O /\ enabled([I,O,H], M) .
```

3. We wrap the right-hand side of any other rule with `normalize` (see the examples).

The following property characterizes the normalized state transition system associated with a `System` term as a *quotient* of the original transition system. It closely resembles classical bisimilarity. An analogous characterization is provided in [13], where a (general) graph canonization is used.

Let t, t', u, u' be (final) terms of sort `System`, $\hat{t}$ a normalized form, r a `System` rule $r: \ s \Rightarrow s'$ and $\hat{r}: \ s \Rightarrow normalize(s')$ (when writing $u \cong t$ we refer to terms that meet Definition 3, in which case $u \cong t$ is equivalent to $\hat{u} = \hat{t}$).

Property 6. *Let rule r meet Definitions 6, 8 and t meet Definition 3.*

- *If $t \overset{r(\sigma)}{\Rightarrow} t'$ then $\exists \sigma' : \hat{t} \overset{\hat{r}(\sigma')}{\Rightarrow} \hat{t}'$ (t' meets Definition 3)*
- *If $\hat{t} \overset{\hat{r}(\sigma)}{\Rightarrow} \hat{t}'$ then $\forall u \cong t \ \exists \sigma', u' \cong t' : \ u \overset{r(\sigma')}{\Rightarrow} u'$ (u' meets Definition 3)*

The poof is merely technical, let us sketch it. The first part comes immediately. If $\hat{t}$ is the normalized form of t then there exists a morphism ϕ which maps the places of t into the places of $\hat{t}$ (by preserving symmetric labelling). Due to the rule's parametric form (Definition 8), if we apply ϕ coherently to the ground substitution σ of r's variables (recall that `Pbag`, `Net` and `System` terms are built of `Place` terms) we induce a ground assignment σ' by which s and s' match $\hat{t}$ and (after normalization) $\hat{t'}$, respectively. The proof of the second part is similar. If $u \cong t$ ($\hat{t}$) there is a morphism ϕ mapping u places to $\hat{t}$ places (and vice-versa). Therefore, given a ground substitution σ in $\hat{r}$ such that $\sigma(s) = \hat{t}$ and $\sigma(s') = \hat{t'}$, we may induce one (coherently with ϕ) that makes s and s' match u and a term u' isomorphic to $\hat{t'}$ (t').

6.5. Specifying production system's graceful degradation through rewrite rules

Fig. 4 depicts a path of the evolution of a system initially composed of two PLs. We may generalize by considering a PT system with N PLs each containing K (two in our example) parallel lines working $K \cdot M$ raw pieces, which maps to the term `NPLsys(N, K, M)`. There are two evolutionary steps:

s1 When a fault affects one of the lines of a PL the PL locally self-adapts to continue working with the left line(s): Its structural evolution is described in detail in Fig. 1 – bottom. The first two transformations depicted in Fig. 4 describe this situation in

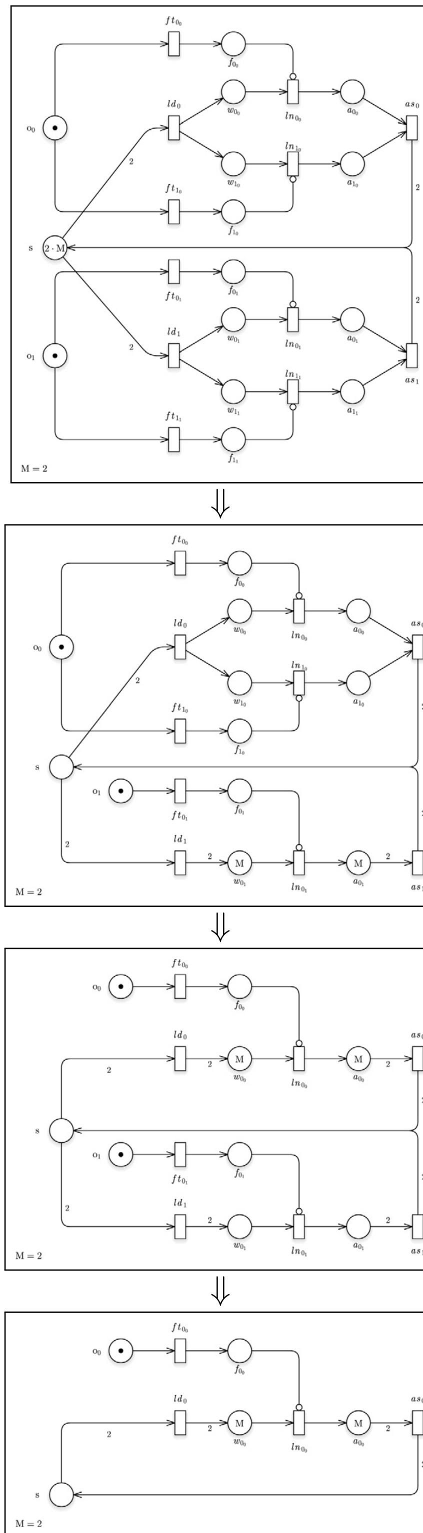


Fig. 4. Gracefully degrading production system with two PLs.

a distributed production system. Contextually, worked pieces on the broken line waiting for assembly need to move to the other(s) line(s) so that the PL can operate in a degraded way. We consider a scenario in which this transformation is delayed until possible, i.e., when the whole system enters a deadlock. The static analysis carried out in Section 5 helps formalize the corresponding rewrite rule.

s2 After a 2nd (K-th) fault affects the (last) working line of a degraded PL, the system self-adapts by detaching the PL (last transformation in Fig. 4): Pieces on the line (some of which may be partially worked) move to the shared warehouse. To add further complexity, we consider a scenario where this transformation is triggered by a *local* deadlock. Again, static analysis helps define the corresponding rule.

We formalize the two transformations as concurrent, non-deterministic rewrite rules (Listing 9). We can verify that both rules are well-defined (Definition 7). Consider, e.g., `r1`: the right-hand side wraps with symmetry-preserving operators the free variable `N'` which is bound in a matching equation to a symmetry-preserving operator taking the variable `N` (the left-hand side's `Net` sub-term) as an argument.

Both `r1` and `r2` exploit `op normalize : System String -> System.` overloading which excludes from index abstraction places with a specific suffix (`"fPL"` and `"PL"`, respectively).

Listing 10. Rewrite rules of the fault-tolerant Production System.

```
*** Legenda "s": warehouse, "a": assembly, op match: sub−bag of places matching a given prefix, op subag: sub−bag matching a given suffix, op |
      |: cardinality
vars S S' S" : Pbag . vars N N' N" : Net . vars Sys Sys' : System .
vars I J : Nat . var W : String . var L : Lab .

*** Replaces a nominal PL with a degraded PL (alias fPL) after a system deadlock
crl [r1] : N S = > normalize(setMark(setMark(attach(N', fPL, "fPL") S" − S',
 "w" "fPL", | match(S', "w") |), "a" "fPL", | match(S', "a") |),"fPL")
   if dead (N S) ∧ S" + 1 . p(< "f" ; J > L < "PL" ; I >) := S ∧
   S' := subag(S", < "PL" ; I >) ∧ N' := detache(N, < "PL" ; I >) .

*** Removes a degraded PL after a 2nd fault and consequent local deadlock
crl [r2] : N S = > normalize(setMark(N" S" − S',"s" ,| match(S,"s") | + | S' |),"PL")
  if S" + 1 . p(< "f" ; J > L < "fPL" ; I >) := S ∧ dead(fPL(I) S) ∧ N" :=
  detache(N, < "fPL" ; I >) ∧ N" =/= emptyN ∧ S' := subag(S", < "fPL" ; I >) .
*** Alternative version of r1 satisfying Definition 6 but not Definition 5
vars Tload Tfail Tfail1 Tfail2 Tass Tline Tline1 Tline2 : String .
vars P0 P1 P2 P3 P4 P5 P6 P7 : Place .
crl [r1.1] : N S + 1 . P7 = > ( N' ; [2 . P1, 2 . P2, nilP] |−> Tload ; [2 . P4, 2 . P1, nilP] |−> Tass ) set(S, P3, 0) + S[P3] . P2 + 1 . P0
 if (N' ; [2 . P1, 1 . P2 + 1 . P3, nilP ] |−> Tload ; [1 . P3, 1 . P5, 1 . P7] |−> Tline2 ; [1 . P4 + 1 . P5, 2 . P1, nilP ] |−> Tass ;[1 . P0,
      1 . P7, nilP] |−> Tfail2 ) := N ∧ dead(N S + 1 . P7) .
```

Rule `r1.1`, instead, is an alternative formalization of step 1 which doesn't meet Definition 6 and therefore cannot be used with normalization.

We will formally prove that the resulting rewritable PT eventually reaches a (safe) final `System` with the `Net` consisting of one component –a non-working degraded PL– and all the $2 \cdot M$ pieces in the warehouse place.

7. Experimental evidences

In this section, we provide evidence of the effectiveness of symmetric net algebra operators for rewritable PT combined with normalization. We aim to show that with this advanced analysis technique, rewritable PT becomes a powerful formalism able to scale up with model complexity. We first consider the model of the distributed Production System alone, and then we include the graceful degradation strategy. We make a comparison with similar approaches.

In this paper, we focus on exact, untimed analysis using the `Maude`'s base model checker. As pointed out in Section 9, there are further extensions under study, in particular, a stochastic version of Rewritable PT.

7.1. Model without graceful degradation

Table 4 refers to the Rewritable PT (PT system) defined by (`NPLsys(N,2,M)`, $R := \{$`firing`$\}$): We test the existence of *final* states in a system composed of `N` replicas of a PL with `K = 2` lines, with `N` 1 through 10. For each `N`, we consider three scenarios: `M = 1`, `M = 2`, and `M = 3` (`2*M` pieces in the warehouse). The search command is: `search NPLsys(N,2,M) =>! F:System .`

The first column refers to an ordinary search graph whereas the second to a normalized one. We observe a dramatic space reduction as `N` grows up. By the way, term normalization induces a computation overhead which, however, is mitigated by the achieved state reduction: This results in execution times magnitude orders below those taken by ordinary searches. For example, for

Table 4
Model of a distributed production system: performance of `search` as `N` (replicas) varies. †Timed out after 2 hours, no solution got with bounded search.

		Ordinary		Normalized	
N	2*M	states(final)	time (sec)	states(final)	time (sec)
1	2	15(2)	0	9(1)	0
	4	42(2)	0	24(1)	0
	6	90(2)	0	50(1)	0
2	2	81(8)	0	17(1)	0
	4	387(12)	0	75(2)	0
	6	1,323(16)	0.1	232(2)	0.2
3	2	351(24)	0	25(1)	0
	4	2,376(48)	0.1	126(2)	0.1
	6	11,232(80)	0.9	498(3)	1
4	2	1,377(64)	0.1	33(1)	0
	4	12,069(160)	1	177(2)	0.3
	6	72,981(320)	10	764(3)	2.6
5	2	5103(160)	0.5	41(1)	0
	4	54,918(480)	7	228(2)	0.6
	6	404,838(1120)	87	1,030(3)	5.4
6	2	18,225(384)	3	49(1)	0.1
	4	232,551(1,344)	51	279(2)	1.1
	6	2,022,975(3,584)	2,133	1,296(3)	9
7	2	63,423(896)	37	57(1)	0.1
	4	936,036(3,584)	793	330(2)	1.7
	6	†	-	1562(3)	14
8	2	216,513(2,048)	116	65(1)	0.2
	4	†	-	381(2)	2.6
	6	†	-	1,828(3)	26
9	2	728,271(4,608)	376	73(1)	0.3
	4	†	-	432(2)	3.3
	6	†	-	2,094(3)	33
10	2	†	-	81(1)	0.4
	4	†	-	483(2)	4.5
	6	†	-	2,360(3)	42

$N = 9$ and $M = 1$ the reduction factor achieved with normalization (and compositional operators) is ≈ 10000 in space and ≈ 1250 in time.

Note that large models can be (exactly) model-checked only using normalization: after two hours, the ordinary searches fail, i.e., do not find any match (final state), even with bounded search depth.

As pointed out previously, another big advantage of normalization is that we more easily reason about the model's behaviour and properties. In this case, the very small number of final states and their being (from a given point on) independent of N help us to give an intuitive interpretation, that otherwise would be impossible.

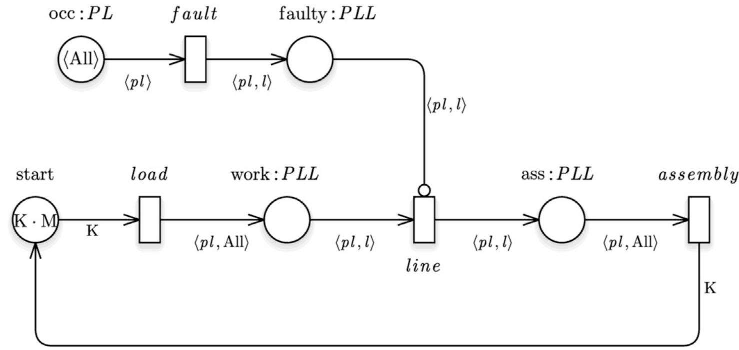
For example, if $N = 1$ or $M = 1$ we get one final state (*up to isomorphism*), coherently with the preliminary analysis carried out in Section 5.1 for $N = 1$. If $N > 1$ the marking sub-term of the final state looks like this (the `Net` sub-term never changes in this model):

```
1 . p(< "a" ; 0 > < "L" ; 0 > < "PL" ; 0 >) +
1 . p(< "w" ; 0 > < "L" ; 1 > < "PL" ; 0 >) +
1 . p(< "f" ; 0 > < "L" ; 1 > < "PL" ; 0 >) +
1 . p(< "f" ; 0 > < "L" ; 0 > < "PL" ; 1 >) +
..
1 . p(< "f" ; 0 > < "L" ; 0 > < "PL" ; N - 1 >)
```

It means that the two pieces stay in a PL (with index 0), one in the faulty line (with index 1) to be worked and the other in the PL second line waiting for assembly. The remaining PLs all have a faulty line (with index 0). The above representation is minimal according to a total order (Section 6.4).

If $N > 1$ and $M = 2$ we get two final states that look like these:

```
Solution 1
2 . p(< "a" ; 0 > < "L" ; 1 > < "PL" ; 0 >) +
2 . p(< "w" ; 0 > < "L" ; 0 > < "PL" ; 0 >) +
1 . p(< "f" ; 0 > < "L" ; 0 > < "PL" ; 0 >) +
1 . p(< "f" ; 0 > < "L" ; 0 > < "PL" ; 1 >) +
.. (N > 2)
1 . p(< "f" ; 0 > < "L" ; 0 > < "PL" ; N - 1 >)
```



class $PL = pl\{1..N\}$ **class** $L = l\{1..K\}$ **domain** $PLL = PL \times L$

var $pl : PL$ **var** $l : L$ $K = 2$ $N = 5$ $M = 2$

Fig. 5. SN model of a production system composed of N PLs each with K lines.

```
Solution 2
 1 . p(< "a" ; 0 > < "L" ; 0 > < "PL" ; 0 >) +
 1 . p(< "w" ; 0 > < "L" ; 1 > < "PL" ; 0 >) +
 1 . p(< "a" ; 0 > < "L" ; 0 > < "PL" ; 1 >) +
 1 . p(< "w" ; 0 > < "L" ; 1 > < "PL" ; 1 >) +
 1 . p(< "f" ; 0 > < "L" ; 1 > < "PL" ; 0 >) +
 1 . p(< "f" ; 0 > < "L" ; 1 > < "PL" ; 1 >) +
 .. (N >2)
 1 . p(< "f" ; 0 > < "L" ; 0 > < "PL" ; N - 1 >)
```

The first match has all four pieces staying in a PL (with index 0), two in the faulty line (with index 0) to be worked and the other two in the PL second line waiting for assembly. The remaining PLs all have a faulty line. The second match shows a different distribution of pieces: two pieces stay in one PL (with index 0) the other two in another PL (with index 1). In each of these PLs, the two pieces are distributed on the inner lines, as described in the previous cases.

We may provide an analogous interpretation for the case $M = 3$, where there are three final states if $N > 2$ (and N final states if $N \leq 2$).

Comparison with related approaches We first compared Rewritable PT enriched with symmetric net operators with related `Maude` approaches: [46] defines reconfigurable PN on the line of algebraic Graph Transformation Systems, i.e., with rewrite rules defined as pushouts. This formalism turned out hard to use and unable to scale model size (it doesn't use any aggregation technique). [38] proposes a general state space reduction technique that exploits a bisimulation relation on states using a canonization function: This technique has turned out cumbersome and inoperable with the hierarchical symmetries pointed out by compositional net operators. Our former canonization technique for Rewritable PT [13], although general (adopting simple indexing of nodes), has proved poorly scalable for models built in a modular way: To give an idea, it takes ≈ 5000 sec to build the canonized search graph for a model equivalent to `NPLsys(4,2,2)` (a system composed of four PLs) against ≈ 0.3 (Table 4).

Symmetric Nets (SN, formerly Well-formed Nets or WN) [17] are a de-facto standard High-Level Petri Net formalism. SN captures system behavioural symmetries thanks to a compact syntax. Several efficient analysis techniques are available for SN, in particular, a quotient reachability graph called Symbolic Reachability Graph or SRG [18].

Fig. 5 shows an SN model equivalent to the term `NPLsys(N, K, M)` (which seemingly is isomorphic to the SN unfolding). SN places (and implicitly transitions) have associated colour domains defined as Cartesian products of finite, disjoint colour classes.[14] SN arcs carry colour functions that preserve the model symmetries (permutations on colour classes).

Table 5 shows the comparison made with the SN solvers integrated into `GreatSPN` graphical editor [1].

The states of ordinary Reachability Graphs, including the final ones (first column), coincide with those of ordinary `search` (first column of Table 4), which confirms the equivalence between models. Interestingly enough, the `Maude` performances are a magnitude order worse than the RG solver despite the wordy terms representing PT systems compared to the compact SN model: this confirms the efficiency of `Maude`'s rewriting engine and of the PT encoding. The second column of Table 5 refers to the Symbolic Reachability Graph of an SN. If we compare it to the second column of Table 4 we note that the performances of the SRG solver are still better than the normalized `search`, but with a significantly lesser ratio than in the ordinary case. This is not surprising, given the *symbolic* firing mechanism adopted by the SRG.

[14] Place *start* in Fig. 5 has neutral domain.

Table 5
Ordinary vs Symbolic Reachability Graph for SN in Fig. 5 (`GreatSPN` solvers). †RG solver crashed or timed out after 2 h: RG size calculated by the SRG solver.

		RG		SRG	
N	2*M	states(final)	time (sec)	states(final)	time (sec)
1	2	15(2)	0	9(1)	0
	4	42(2)	0	24(1)	0
	6	90(2)	0	50(1)	0
2	2	81(8)	0	23(2)	0
	4	387(12)	0	108(4)	0
	6	1,323(16)	0	347(4)	0
3	2	351(24)	0	44(3)	0
	4	2,376(48)	0	249(6)	0
	6	11,232(80)	0	1,075(9)	0
4	2	1,377(64)	0	71(4)	0
	4	12,069(160)	1	451(9)	0
	6	72,981(320)	5	2,226(14)	1
5	2	5103(160)	0	105(5)	0
	4	54,918(480)	3	710(11)	0
	6	404,838(1,120)	33	3,808(19)	1
6	2	18,225(384)	1	145(6)	0
	4	232,551(1,344)	17	1,030(14)	1
	6	2,022,975(3,584)	207	5,813(24)	2
7	2	63,423(896)	4	192(7)	0
	4	936,036(3,584)	84	1,407(16)	1
	6	9,386,604(10,752)†	-	8,249(29)	3
8	2	216,513(2,048)	17	245(8)	0
	4	3,628,233(9,216)	382	18,45(19)	1
	6	41,178,921(30,720)†	-	11,108(34)	4
9	2	728,271(4,608)	64	305(9)	0
	4	13,660,002(23,040)	1,835	2,340(21)	2
	6	173,328,498(84,480)†	-	14,398(39)	6
10	2	2,410,769(10,240)	250	371(10)	0
	4	50,250,699(56,320)†	-	2,896(24)	3
	6	704,513,619(225,280)†	-	18,111(44)	8

Enhanced effectiveness What is most noticeable, however, is that the state reduction achieved through the normalization technique integrated into `Maude` is more effective than the reduction achieved through the SRG. As N grows the reduction rate becomes significant. This behaviour, surprising in part, has a logical explanation: Our normalisation method uses a coarser equivalence on states than the SRG. The symmetries caught by SN are only horizontal (permutations on colour classes), instead, those captured by symmetrical labelling of `Nets` are both horizontal and vertical (they reflect the model's hierarchical definition). Consider, e.g., `NPLsys(2,2,1)`. The `search` for final states has one match (Table 4, 2nd col, 4th row) against the two final states of the SRG (Table 5, same coordinates). The marking sub-term of the `search` solution is:

```
1 . p(< "a" ; 0 > < "L" ; 0 > < "PL" ; 0 >) +
1 . p(< "w" ; 0 > < "L" ; 1 > < "PL" ; 0 >) +
1 . p(< "f" ; 0 > < "L" ; 1 > < "PL" ; 0 >) +
1 . p(< "f" ; 0 > < "L" ; 0 > < "PL" ; 1 >)
```

The SN model (Fig. 5) uses two colour classes, one (PL) for production units and the other (L) for inner lines, of size N and K, respectively. SRG nodes (called Symbolic Markings) represent equivalence classes using symbols denoting *parametric partitions* of colour classes. The Symbolic Marking corresponding to the `Pbag` term above looks like this (we use intuitive notation where symbols $\{z^i_C\}$ represent a partition of class C, in this example $|z^i_C| = 1$ for each C, i):

$$m(ass) := \langle z^0_{PL}, z^0_L \rangle \;\; m(work) := \langle z^0_{PL}, z^1_L \rangle \;\; m(faulty) := \langle z^0_{PL}, z^1_L \rangle + \langle z^1_{PL}, z^0_L \rangle$$

This SM represents any final state where the faulty lines in each of the two PLs have different indices. It has a counterpart (in SN) where they take the same index. Since normalization works on different nesting levels (of place labels) independently it identifies these two states.

The different aggregation power is remarkable when the number of lines in a PL (the parameter K) is bigger. Table 6, e.g., refers to $K = 3$, with 9 ($M = 3$) pieces worked at each production cycle. Again, we are searching for final states using: `search NPLsys(N,3,3) =>! D:System` . For example, for $N = 10$ the reduction factor is ≈ 45. Moreover, with normalization the number of final states remains constant ($= 3$) for $N \geq 3$: This is crucial if we want to easily test system properties like distributed termination and is a symptom of structural behaviour. Finally, we observe that the difference between execution times tends to subside as the model's size increases.

The ordinary search graphs have huge sizes (they are calculated by the SRG solver): For $N = 9$ and $N = 10$ more than 16 billion states (with 3,247,695 final states) and more than 88 billion (with 12,990,780 final states), respectively.

Table 6
SRG vs Rewritable PT normalization ($K = 3$, $M = 3$, 9 pieces worked).

N	SRG (SN)		Normalized state space (Maude)	
	states(final)	time (sec)	states(final)	time (sec)
1	100(1)	0	100(1)	0
2	1,092(4)	0	534(2)	3.7
3	4,747(12)	2	1,188(3)	18
4	12,748(22)	6	1,842(3)	42
5	26,865(37)	15	2,496(3)	77
6	48,787(55)	30	3,150(3)	103
7	80,278(77)	60	3,804(3)	127
8	123,027(102)	89	4,458(3)	163
9	178,804(132)	137	5,112(3)	214
10	249,292(164)	211	5,766(3)	290

Table 7
Model including graceful degradation: Performance of `search` command as `N` (replicas) varies. †Timed out after 2 hours, no solution got with a bounded search.

N	2*M	Ordinary		Normalized	
		states(final)	time (sec)	states(final)	time (sec)
1	2	23(2)	0	17(2)	0
	4	60(2)	0	42(2)	0
	6	122(2)	0	82(2)	0
2	2	169(4)	0.02	50(2)	0
	4	645(4)	0.07	172(2)	0
	6	1915(4)	0.1	451(2)	0.3
3	2	944(6)	0.1	97(2)	0
	4	4,579(6)	0.4	373(2)	0.2
	6	17,463(6)	1.8	1,153(2)	1.3
4	2	4,764(8)	0.7	158(2)	0
	4	27,316(8)	3.6	642(2)	0.6
	6	124,148(8)	21	2,123(2)	3.2
5	2	22,934(10)	4.5	233(2)	0.1
	4	149,085(10)	35	979(2)	1
	6	769,869(10)	434	3,357(2)	5.8
6	2	107,744(12)	42	322(2)	0.1
	4	774,470(12)	690	1,384(2)	1.8
	6	†	-	4,855(2)	13
7	2	499,166(14)	532	425(2)	0.5
	4	†	-	1,857(2)	3.3
	6	†	-	6,617(2)	22
8	2	†	-	542(2)	0.75
	4	†	-	2,398(2)	5.2
	6	†	-	8,643(2)	37
9	2	†	-	673(2)	1.1
	4	†	-	3,007(2)	8
	6	†	-	10,933(2)	55
10	2	†	-	818(2)	2
	4	†	-	3,684(2)	13
	6	†	-	13,487(2)	77

7.2. Including graceful degradation

We are principally interested in the analysis of the model including the specification of the distributed system's graceful degradation: Formally, the Rewritable PT defined by (`NPLsys(N,2,M)`, $R := \{\texttt{firing}, \texttt{r1}, \texttt{r2}\}$).

The data in Table 7 refers to this model. We formally prove that we eventually reach a final `System` term with the `Net` sub-term consisting of one degraded PL holding the $2 \cdot M$ pieces (all to work except at most one). In the normalized case, this corresponds to the following command[15]:

```
search NPLsys(N,2,M) =>! F:System such that
net(F:System) == faultyPL /\ M:Pbag := marking(F:System) /\
| match(M:Pbag, "w") | + | match(M:Pbag, "a") | == 2 * M  .
```

[15] In the ordinary case the command is a bit more complex.

Table 8
SN-based PT emulator vs modular Rewritable PT.

N	2*M	SN-based emulator (SRG)		RewPT (Normalized)	
		states (obs. + van.)	time (sec)	states(final)	time (sec)
1	2	17(2)+3,277	1.4	17(2)	0
	4	42(2)+12,706	10	42(2)	0
	6	82(2)+50,114	55	82(2)	0
2	2	68(2)+51,454	57	50(2)	0
	4	251(2)+178,378	167	172(2)	0
	6	668(2)+598,407	976	451(2)	0.3

For each N, M this command has two matches as the simpler one:

```
search NPLsys(N,2,M) =>! F:System .
```

The outcome is coherent with the analysis carried out in Section 5.

If we compare Table 7 to Table 4 we note that the system state space is considerably larger because it includes the system's adaptation for gracefully degrading. We can make analogous considerations to Section 7.1. With normalization, we achieve impressive state space reduction: e.g., for $N = 7$, $M = 1$ the reduction ratio is ≈ 1100, pretty much the same as in Table 4 for the same values. The model's configurations solvable with an ordinary depth-bounded search (with a two-hour timeout) are fewer than in the scenario without graceful degradation.

Comparison with related approaches When considering graceful degradation, the alternative `Maude` approaches mentioned in the previous sub-section are hardly feasible. The same can be said for Symmetric Nets: Getting out a corresponding model from that in Fig. 5 (which, however, required some expertise) is a sort of "mission impossible": In particular, encoding the conditions that trigger the system's evolution.

The only comparison we can make with SN is through the framework we have presented in [12]: a large SN emulates any PT system, which is encoded as an SN marking. The model is completed by an API of base PT transformations, defined in turn as SNs. Due to the low abstraction provided by SN colouring, we have to use lots of *immediate* transitions which produce a large number of *vanishing* (i.e., non-observable) states.

This framework hardly scales the size of models built in a modular/hierarchical way: Table 8 shows the performances of the SN-based emulator against modular Rewritable PT for small model sizes. Data refer to the SRG and normalized search graph, respectively (these are the same as in Table 7): The massive number of invisible states and the encoding of PT nets as marking (with consequent overhead in symbolic marking canonization) considerably affect the SN-based PT emulator. We believe that by using structural analysis to reduce immediate transitions interleaving we might alleviate in part this evident gap.

8. Marking abstraction: `Net` projection

We conclude the part on rewritable PT analysis by discussing a possible alternative/complement to exact state space search.

The idea is to apply state abstraction to get out the *potential* evolution of a system's topology, from which (under finiteness hypothesis) one can test structural properties, e.g., by computing the semi-flows of reachable `Nets`.

Similarly to what is done in [34] for net-token markings in nets-within-nets, we are interested in the *projection* on `Net` sub-terms of reachable `System` terms of rewritable PT. This concept has intuitive formalization.

Definition 9 *(`Net` projection).* Let $TS(\texttt{net m0}, R) := (V, E)$ be the state-transition system of `net m0` (Definition 1). The `Net` projection $\Pi_{\texttt{Net}}(\texttt{net m0}, R)$ is a graph (V', E') whose nodes are `Net` ground terms such that:

$\texttt{n} \in V'$ if and only if $\exists\, \texttt{(n m)} \in V$,

$\texttt{(n,n')} \in E'$ if and only if $\exists\, \texttt{(n m)} \overset{r(\sigma)}{\Rightarrow} \texttt{(n' m')} \in E$.

Due to the possible huge (or even infinite) size of TS, we try to directly build an *over-approximation* of $\Pi_{\texttt{Net}}(TS)$ from a rewritable PT rather than trying to derive it from the associated TS. The smaller, the better.

A classical approach used in `Maude` to reduce the complexity of state space search is to introduce state abstraction through extra equations. In our context, a naive way of going in this direction would be to add the equation:

```
eq N:Net M:Pbag = N:Net nilP.
```

which identifies `System` terms with the same marking. A major drawback of this simple solution is that it likely results in under-approximations of the projection: Rewrite rules, indeed, are usually triggered by conditions on markings.

The solution we propose consists of a kind of syntactical rewriting of `System` rules into abstract versions in which the rule's terms on both sides have a null marking (`nilP` represents the empty bag) so that: if $r \in R - \{\texttt{firing}\}$ and r' is the abstraction of r then for each `System` ground term `n m`:

if $\texttt{n m} \overset{r}{\Rightarrow} \texttt{n' m'}$ then $\texttt{n nilP} \overset{r'}{\Rightarrow} \texttt{n' nilP}$.

Table 9
Marking abstraction of the gracefully degrading system model: synthesis of three `search` commands as `N` (replicas) varies.

N	Ordinary		Normalized	
	states(final)	time (sec)	states(final)	time (sec)
1	2(1)	0	2(1)	0
2	8(2)	0	5(1)	0
3	26(3)	0	9(1)	0
4	80(4)	0.2	14(1)	0
5	242(5)	0.7	20(1)	0
6	728(6)	2.5	27(1)	0.2
7	2,186(7)	6.5	35(1)	0.3
8	6,560(8)	34	44(1)	0.6
9	19,682(9)	151	54(1)	0.9
10	59,048(10)	875	65(1)	1.5

The technique we are going to present is currently semi-automatic: One step (the fourth) may require some heuristics. Let us sketch it and show an application to the case study.

1. We replace the marking (`Pbag`) sub-term of the left-hand side of r as well as any other references to it with `nilP` (also in rule's condition, if any).
2. We replace each operator (`sysop`) of range `System` with its abstract version (`sysopAbs`) defined as `eq sysopAbs(..) = net(sysop(..)) nilP .` such that if `sysop(..)`→ `n m` then `sysopAbs(t,t',..)`→ `n nilP`. If the operator `sysop` retains the `Net` sub-term of its `System` argument we can simply unwrap the latter (see the example below).
3. If r is conditioned, we replace each predicate/matching equation that refers directly or indirectly to the marking with `true` (or equivalently, we erase it).
4. Possible free variables resulting from marking abstraction have to be bound to the left-hand term: For example, if a free variable refers directly or indirectly (in the original rule) to a net `Place` we may add a matching equation setting its membership to the `Net` sub-term.

Below is the marking abstraction of the two rewrite rules of the running example (see Listing 9, the `firing` rule obviously disappears). Since the `setMark` operator doesn't change the `Net` sub-term of its `System` argument we may (recursively) unwrap the latter instead of using `setMarkAbs`. We add the first clause on rule conditions to bind the free variable `I` to a place of the `Net`.

Listing 11. Marking abstraction of rewrite rules of the FT Production System.

```
vars N N' N" : Net .
vars I J : Nat .
vars L L' : Lab .
*** Replaces a nominal PL with a degraded PL (alias fPL)
crl [ar1] : N nilP => normalize(attach(N', faultyPL, "fPL") nilP)
    if p(L < "PL" ; I >) U Sp := places(N) /\ N' := detache(N, < "PL" ; I >) .

*** Removes a degraded PL
crl [ar2p] : N nilP => normalize(N" nilP) if p(L < "fPL" ; I >) U Sp := places(N) /\ N" := detache(N, < "fPL" ; I >) /\ N" =/=
     emptyN .
```

Table 9 collects the outcomes of three different searches on the marking abstraction of the rewritable PT specification of the distributed gracefully degrading production system. The first command shows all reachable states of a system (* means zero or more steps): In this case, how the system's topology may evolve. The second is an ordinary search for final states. The third command does the same but with the additional constraint that the final state(s) match(es) a degraded PL: It makes sense only when we carry out normalization.

```
search NPL(N,2) nilP =>* F:System .
search NPL(N,2) nilP =>! F:System .
search NPL(N,2) nilP =>! F:System such that
net(F:System) == faultyPL .
```

The three searches give coherent results. Looking at the second column, which refers to normalized states,[16] we may assert to have formally proved that the gracefully degrading system eventually reaches a final configuration with only one degraded PL. Fig. 6,

[16] Normalization boils down to index abstraction.

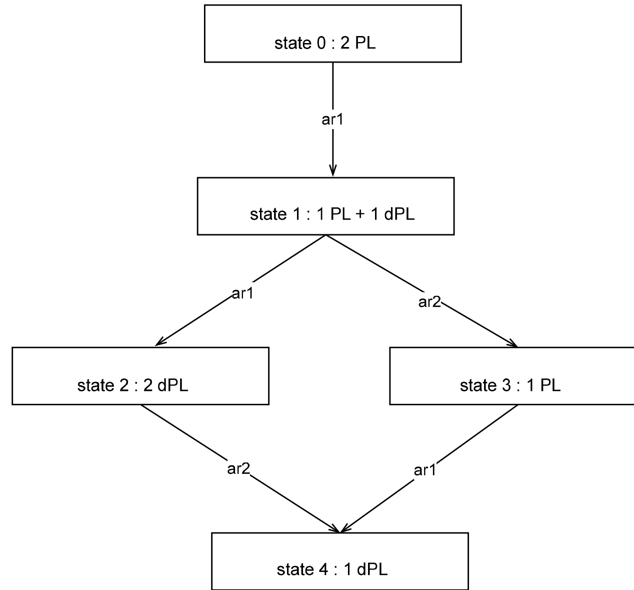


Fig. 6. Degradation of a system with 2 PL (Net projection).

directly derived from the `show search graph .` command, illustrates the normalized state transition system of `NPL(2,2) nilP`. using marking abstraction: PL and dPL denote a nominal and degraded production unit, respectively.

9. Related work (continued)

In the introduction, we have mentioned some of the available formalisms for modelling and (with severe limitations) verifying dynamically reconfigurable distributed systems, in particular, higher-order Petri Nets (Nets-within-Nets). In Sections 7.1 and 7.2 we have compared the (exact) model-checking capability of modular Rewritable PT with those of related approaches (based on `Maude` [46], [38], [13] and Symmetric Nets [18], [12]) and shown (for the running example) that these alternatives either are not viable or do not scale the model size.

Process calculi (or Algebras) [4], [42], [25] aim at describing formally the behaviour of concurrent systems in a modular way. Variations on classical definitions capture the dynamic character of concurrency, e.g., the π-calculus [43] describes mobile systems in which communication topology changes dynamically. The Ambient calculus [16] builds on π-calculus mobility to describe the dynamics of interaction within boundaries and hierarchies. In these calculi, the dynamic behaviour of a system consists of local changes. More recent proposals, e.g., [7], [20] (just to mention a few) include constructs to represent adaptation patterns at the process level and describe process evolution over time.

For finite-state models, reachability analysis is the most simple and complete technique. Two solutions have been studied for a long time to fight its possible combinatorial complexity (not only for process calculi but for any formalism for concurrent systems). Compositional analysis tries to derive properties of a system starting from those of its components: A foundational work for process calculi is [24]. Conversely, Equivalence check (or symmetry detection) consists of finding bisimulations or simulation preorders in labelled transition systems (and lumpability conditions for their stochastic extensions into Markov Chains). [55] provides an exhaustive overview of equivalence checking (mostly) and compositional verification for the leading formalisms for concurrent systems.

Compositional approaches have shown very low practical usability because of the severe restrictions on model structure. The approaches based on equivalence checking or symmetry detection have greater applicability but do not always scale the model size. More importantly, they usually work with classical formalisms, not including dynamic features.

Symmetry reduction techniques, including canonical forms, have been studied for a long time in the Petri Net area: [50], [27], [49], [29] (the last applies to PT markings). The most mature and popular outcome is Symmetric Nets (SN, formerly Well -formed Nets or WN) [17], a de-facto standard High-Level Petri Net formalism: SN captures system behavioural symmetries (corresponding to colour permutations) thanks to a compact syntax. Efficient analysis techniques are available for SN, in particular, a quotient reachability graph called Symbolic Reachability Graph or SRG [18]. To our knowledge, equivalence check or symmetry reduction only applies to classical (low- or high-level) PN classes.

As for `Maude` and rewriting logic, we have to mention [38], which proposes a general state space reduction technique that exploits a bisimulation relation on states using a canonization function. This technique has turned out cumbersome (in the symmetry specification part) and inoperable with the modular symmetry structure pointed out by our compositional net operators.

Rewritable PT nets may be seen as instances of Graph Transformation Systems. Let us finally mention some well-known algorithms for graph canonization (GC). It comes out that there is a disparity between the theory and practice of handling GC (and graph

isomorphism): polynomial algorithms exist only for sub-classes of graphs. A complete survey of the existing algorithms/tools is available in [44]. Among the best-known algorithms, only Bliss [28] and Nauty/Traces [39] get a canonized form for graphs with labelled vertices (like our nets). They handle the GC problem by finding an equitable colouring of vertices, from which they induce an ordering with the help of search trees using backtracking. Progressive pruning permits a reduction of the solution space. When edges are labelled, encoding into graphs with unlabelled edges is usually necessary. The approach of Scott [5] works on generally labelled graphs by translating them reversibly into trees (simpler to canonize): It seemingly performs a bit slower than the state-of-art tools. The canonization technique for Rewritable PT [13] is inspired by GC algorithms. Different from [13], the normalization technique used in this paper (exploiting compositional operators) makes deterministic choices and doesn't need backtracking to get the minimal form for a PT system.

10. Strength and weakness: a short discussion

In this section, we sketch the strengths and weaknesses of `Maude` modular RwPT, based on our practical experience.

Scalability, usability The formalism has been successfully employed to specify and analyse large-scale models of distributed dynamic systems. It outperforms related approaches. Compositional operators are simple to use and cover most modelling needs.

Availability of tools The `Maude` ecosystem provides model-checking facilities (we have used the basic one) and other tools for model validation. Using a PN-based formalism makes it possible to inter-operate with other tools like `GreatSPN` for checking, e.g., structural properties.

Extensibility The base version of RwPT may be easily enriched with new features, for example, transition or rule priorities (and time attributes). The simple implementation of normalization promotes the integration of more sophisticated heuristics.

Verbosity As usual in algebraic specifications, terms specifying PT systems may be wordy. This moderately affects the rewriting efficiency and, to a greater extent, readability and interpretation of analysis outcomes.

High sensitivity As usual for systems with rewriting semantics, (time) efficiency of reachability analysis strictly depends on how operations/rewrite rules are defined: Functionally equivalent definitions may have diverging performances. This requires expertise and or accurate debugging/tuning.

Limitations Currently, we can formally verify untimed, finite-state models. The ability to extract performance or dependability metrics from distributed system models would be highly desirable.

11. Conclusion and future work

This paper has presented a *compositional* modelling approach for adaptive distributed systems based on a `Maude` encoding of "Rewritable" PT nets. Net-algebra operators point out (through structured node labelling) symmetries present in the model's modular hierarchy. Intercession operators allow one to specify rewrite rules preserving the model's symmetry structure. An effective state space normalization technique (fully integrated into `Maude`) exploits the symmetry hierarchy to get a compact state transition system which describes the model's behaviour.

The proposed (place-oriented) formalization of rewritable PT nets simplifies a previous one by making it more effective and suitable for modular state normalization. The formal specification of a gracefully degrading distributed system has been used throughout the paper as a benchmark for model scalability. Experimental evidence has proven that Rewritable PT enriched with compositional operators is a powerful formalism able to scale system size (in terms of nested components) and outperforms related approaches based on `Maude` or Coloured Petri Nets.

We have also sketched the potential advantages of integrating state space analysis with structural techniques (typical of Petri nets) and the use of state abstraction to get a simplified, net-oriented view of system evolution.

We end our contribution with a non-exhaustive list of future/ongoing developments.

Optimizations to normalization: normalized transition firing Besides the simple heuristics already implemented, there are a few structural optimizations that might further impact efficiency. The normalization technique currently applies to states (nets plus markings), not to transition firings,[17] as in SN. Using the bisimilarity between isomorphic states we might reduce firing instances as well by selecting a representative for a whole class leading to the same (normalized) state. This would make the approach closer to a symbolic one.

Extending symmetries The symmetry hierarchy currently captured through compositional net operators builds on (local) index permutations. From the modelling point of view (e.g., to describe circular topology), it could help to represent more specific symmetries like rotations (permutations modulo-n).

[17] Apart from reductions indirectly achieved through state reduction.

Integration of structural techniques in `Maude` As discussed, this is a potential opportunity. A short-term goal is to extend PN structural relations (conflict, causality, mutual exclusion) to RwPT: Not only between instances of `firing` rule (as in classical PN) but also between different rewrite rules. One interesting application could be the use of partial order (state space) techniques like the Stubborn Set method [53] for RwPT models.

Timed analysis In this paper, we have used `Maude` model-checking facilities to formally verify the graceful degradation of a distributed system into a proper final state. More interesting (performance, dependability) properties take time into account: For example, the system's (Mean) Time To Failure and (average) Throughput. Using richer annotations for "transitions" and rules, including (exponential) rates, we may easily extend the RwPT formalism to derive a lumped Markov chain [9] directly from the RwPT normalized state-transition system: the Bisimilarity condition indeed maps to (exact or ordinary) lumpability. The computation of transition rates between aggregates from the transition system's edges is a simple enumeration. If transition firings were normalized we should compute (using combinatorics) their cardinality.

Fighting verbosity As we have pointed out, the wordy terms specifying PT systems slightly affect the efficiency but even more the model's readability and the interpretation of analysis results. There are at least two ways to face verbosity. The simpler one is to extend the naive aliasing employed with a term-unfolding mechanism: The idea is to leave the representation of a modular PT system implicit and unfold it when needed. A more systematic solution would be encoding High-level PNs in `Maude`. Despite some naive proposals, this requires an extensive study (e.g., to evaluate the viability of `Maude`'s reflective ability).

Infinite-state analysis We have considered (huge) finite-state models. What if we need to specify an infinite model? A technique largely used in PN area to face unbounded markings is coverability analysis [22]. Parametrized PN analysis [23] instead consists of checking properties (e.g., reachability) for a Petri net whose initial marking contains some parameter (e.g., it could be M for our case study). The ability to use these techniques for RwPT, in which the PT system topology varies over time, deserves attention.

Extension to Net-within-Nets `Maude` provides a natural way to enrich a formalism like PN with dynamic features. We have recently encoded the base class of Nets-within-Nets (Elementary Object Systems, or EOS) in `Maude` [15]. Some of the ideas discussed in this paper might successfully apply to EOS: E.g., we may make the system PT net (holding net-tokens) rewritable in turn and normalize net-tokens built in a modular way.

Writing models efficiently As any language with rewriting semantics, `Maude` is simple to use but requires expertise in defining operations and rules efficiently. In our context, we may alleviate this (general) performance issue by reducing the modeller task as far as possible: For example, by further extending the library of net-intercession operators. As a long-term project we like to assist the modeller by tool-support within a graphical user interface. Here, we like to integrate the presented approach into the RENEW-Tool [36] as a plug-in.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. The core PT-NET `Maude` module

Listing 12. `Maude` specification of PT nets.

```
*** New, effective PT net signature: a net is a map Tmatrix −> Tag
fmod PT−NET{X :: TRIV} is
 pr MAP−SET{Tmatrix,X} * (sort Map{Tmatrix,X} to Net, sort Entry{Tmatrix,X} to Transition, op emptyM to emptyN, op emptyS to
      emptyTset) .
 pr SET+{X} * (op emptyS to emptyStlab) .

 vars N N' N'' : Net .
 vars T T' : Transition .
 vars P P' P'' : Place .
 vars Y J : Nat .
 vars K K' : NzNat .
 vars I O H B : Pbag .
 vars Q Q' : Tmatrix .
 vars S S' PS : Pset .
 var W : String .
 var WL : List{String} .
 var NWL : NeList{String} .
 var Lab : Lab .
```

```
var NeL : NeLab .
var L : X$Elt . *** tran. lab
var Z : Set{X} .
*** transition label
op lab : Transition -> X$Elt .
eq lab(Q |-> L) = L .
*** dead transition
op dead : Transition -> Bool .
eq dead(Q |-> L) = dead(Q) .
*** test the existence of a place
op in : Net Place -> Bool .
ceq in(Q |-> L ; N, P) = true if in (Q, P) .
eq in(N, P) = false [owise] .
*** test the existence of a transition
op in : Net Tmatrix -> Bool .
eq in(N, Q) = $hasMapping(N, Q) .
*** get out the places of the net
op places : Net -> Pset [memo] .
eq places(N) = $places(N, emptyPset) .
op $places : Net Pset -> Pset .
eq $places(emptyN, S) = S .
eq $places(Q |-> L ; N, S) = $places(N, S U places(Q)) .
*** extract the subnet matching a given suffix (a list of tags)
op subnet : Net NeList{String} -> Net .
eq subnet(N, NWL) = $subnet(N, NWL, emptyN) .
op $subnet : Net NeList{String} Net -> Net .
eq $subnet(emptyN, NWL, N) = clearup(N) .
eq $subnet(Q |-> L ; N, NWL, N') = $subnet(N, NWL, N' ; sumatrix(Q, NWL) |-> L) .
*** overloaded version: part of the suffix tail is specified exactly
op subnet : Net List{String} NeLab -> Net .
eq subnet(N, WL, NeL) = $subnet(N, WL, NeL, emptyN) .
op $subnet : Net List{String} NeLab Net -> Net .
eq $subnet(emptyN, WL, NeL, N) = clearup(N) .
eq $subnet(Q |-> L ; N, WL, NeL, N') = $subnet(N, WL, NeL, N' ; sumatrix(Q, WL, NeL) |-> L) .
*** overloaded version: the suffix tail is specified exactly
op subnet : Net NeLab -> Net .
eq subnet(N, NeL) = subnet(N, emptyLs, NeL) .
*** extract the subnet matching a given pre-suffix
op netmatch : Net List{String} NeLab -> Net .
eq netmatch(N, WL, NeL) = $netmatch(N, WL, NeL, emptyN) .
op $netmatch : Net List{String} NeLab Net -> Net .
eq $netmatch(emptyN, WL, NeL, N) = clearup(N) .
eq $netmatch(Q |-> L ; N, WL, NeL, N') = $netmatch(N, WL, NeL, N' ; match(Q, WL, NeL) |-> L) .
*** places matching a suffix (derived definitions)
op places : Net NeList{String} -> Pset .
eq places(N, NWL) = places(subnet(N, NWL)) .
op places : Net List{String} NeLab -> Pset .
eq places(N, WL, NeL) = places(subnet(N, WL, NeL)) .
op placematch : Net List{String} NeLab -> Pset .
eq placematch(N, WL, NeL) = places(netmatch(N, WL, NeL)) .
*** removes self-loops
op clearup : Net -> Net .
eq clearup([I, I, H] |-> L ; N) = clearup(N) .
eq clearup(N) = N [owise] .
*** removes the 2nd net from the 1st, if included (does nothing otherwise)
op detache : Net Net -> Net .
eq detache(N ; N', N') = N .
eq detache(N, N') = N [owise] .
*** symmetric version used in the paper
op detache : Net Elab List{String} -> Net .
op detache : Net Elab -> Net .
eq detache(N, El:Elab) = detache(N, El:Elab, emptyLs) .
*** remove net places
op remove : Net Pset -> Net .
eq remove(N, emptyPset) = N .
eq remove(N, S) = $remove(N, emptyN, S) .
```

```
 op $remove : Net Net NePset -> Net .
 eq $remove(emptyN, N, S) = N .
 eq $remove(Q |-> L ; N', N, S) = $remove(N', remove(Q, S) |-> L ; N, S).
*** remove transitions with given tags
 op remove : Net Set{X} -> Net .
 eq remove(N, emptyStlab) = N .
 eq remove(N, Z) = $remove(N, emptyN, Z) .
 op $remove : Net Net NeSet{X} -> Net .
 eq $remove(emptyN, N, Z) = N .
ceq $remove(Q |-> L ; N', N, Z) = $remove(N', N, Z) if L in Z .
 eq $remove(Q |-> L ; N', N, Z) = $remove(N', Q |-> L ; N, Z) [owise] .
 *** net-algebra operators
 *** add a new tag to the net's places, excluding those to share
 op addLab : Net Lab Pset -> Net .
 eq addLab(N, Lab, S) = $addLab(N, emptyN, Lab, S) .
 op $addLab : Net Net Lab Pset -> Net .
 eq $addLab(emptyN, N, Lab, S) = N .
 eq $addLab(Q |-> L ; N', N, Lab, S) = $addLab(N', addLab(Q, Lab, S) |-> L ; N, Lab, S) .
 *** simple (default) version
 op addLab : Net Lab -> Net .
 eq addLab(N, Lab) = addLab(N, Lab, emptyPset) .
 *** defined for convenience (maps to the previous version)
 op addLab : Net Lab List{String} -> Net .
 eq addLab(N, Lab, WL) = addLab(N, Lab, places(N, WL))
 *** replace the suffix in all places matching the given pattern with a similar one whose header's index has the specified value
 op replaceWith : Net NeLab Nat -> Net .
 eq replaceWith(N, NeL, Y) = $replaceWith(N, NeL, Y, emptyN) .
 op $replaceWith : Net NeLab Nat Net -> Net .
 eq $replaceWith(emptyN, NeL, Y, N) = N .
 eq $replaceWith(Q |-> L ; N, NeL, Y, N') = $replaceWith(N, NeL, Y, N' ; replaceWith(Q, NeL, Y) |-> L) .
endfm
```

Appendix B. How to run the example

- Download core Maude and install it (follow the instructions)[18]
  http://maude.cs.illinois.edu/w/index.php/The_Maude_System#Obtaining_and_Using_Maude
- Download the source codes (better downloading the whole repository)
  https://github.com/lgcapra/rewpt
- Launch the Maude shell, e.g., with

```
~$ ./Linux64/maude.linux64
```

- Type

```
Maude> cd rewpt/modPT
Maude> in FT-PL
```

- Search example: type

```
Maude> search NPLsys(5,2,2) =>! F:System .
```

References

[1] E.G. Amparore, G. Balbo, M. Beccuti, S. Donatelli, G. Franceschinis, 30 years of GreatSPN, in: Principles of Performance and Reliability Modeling and Evaluation, Springer, 2016, pp. 227–254.

[2] L. Babai, Graph isomorphism in quasipolynomial time [extended abstract], in: Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, NY, USA, 2016, pp. 684–697.

[18] In our experiments we have used release 3.2.2.

[3] P.E.S. Barbosa, J.P. Barros, F. Ramalho, L. Gomes, J. Figueiredo, F. Moutinho, A. Costa, A. Aranha, SysVeritas: a framework for verifying IOPT nets and execution semantics within embedded systems design, in: L.M. Camarinha-Matos (Ed.), Technological Innovation for Sustainability - Second IFIP WG 5.5/SOCOLNET Doctoral Conference on Computing, Electrical and Industrial Systems, DoCEIS 2011, Costa de Caparica, Portugal, February 21-23, 2011. Proceedings, Springer, 2011, pp. 256–265.
[4] J.A. Bergstra, J.W. Klop, Process algebra for synchronous communication, Inf. Control 60 (1984) 109–137.
[5] N. Bloyet, P.F. Marteau, E. Frenod, Scott: a method for representing graphs as rooted trees for graph canonization, in: COMPLEX NETWORKS 2019, in: Studies in Computational Intelligence Series, Springer, 2019, pp. 578–590, https://hal.archives-ouvertes.fr/hal-02314658.
[6] A. Bouhoula, J.P. Jouannaud, J. Meseguer, Specification and proof in membership equational logic, Theor. Comput. Sci. 236 (2000) 35–132, https://doi.org/10.1016/S0304-3975(99)00206-6.
[7] M. Bravetti, C. Di Giusto, J.A. Pérez, G. Zavattaro, Adaptable processes, Log. Methods Comput. Sci. 8 (2012).
[8] R. Bruni, J. Meseguer, Generalized rewrite theories, in: J.C.M. Baeten, J.K. Lenstra, J. Parrow, G.J. Woeginger (Eds.), Automata, Languages and Programming, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 252–266.
[9] P. Buchholz, Exact and ordinary lumpability in finite Markov chains, J. Appl. Probab. 31 (1994) 59–75, http://www.jstor.org/stable/3215235.
[10] R.M. Burstall, J.A. Goguen, Algebras, theories and freeness: an introduction for computer scientists, in: M. Broy, G. Schmidt (Eds.), Theoretical Foundations of Programming Methodology: Lecture Notes of an International Summer School, directed by F. L. Bauer, E. W. Dijkstra and C. A. R. Hoare, Springer Netherlands, Dordrecht, 1982, pp. 329–349.
[11] L. Cabac, M. Duvigneau, D. Moldt, H. Rölke, Modeling dynamic architectures using nets-within-nets, in: Proceedings of the 26th International Conference on Applications and Theory of Petri Nets, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 148–167.
[12] M. Camilli, L. Capra, Formal specification and verification of decentralized self-adaptive systems using symmetric nets, Discrete Event Dyn. Syst. (2021), https://doi.org/10.1007/s10626-021-00343-3.
[13] L. Capra, Canonization of reconfigurable PT nets in Maude, in: A.W. Lin, G. Zetzsche, I. Potapov (Eds.), Reachability Problems, Springer International Publishing, Cham, 2022, pp. 160–177.
[14] L. Capra, Rewriting logic and Petri nets: a natural model for reconfigurable distributed systems, in: R. Bapi, S. Kulkarni, S. Mohalik, S. Peri (Eds.), Distributed Computing and Intelligent Technology, Springer International Pub., Cham, 2022, pp. 140–156.
[15] L. Capra, M. Köhler-Bussmeier, Modelling adaptive systems with nets-within-nets in Maude, in: Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE, INSTICC, SciTePress, 2023, pp. 487–496.
[16] L. Cardelli, A. Gordon, Mobile ambients, Theor. Comput. Sci. 240 (2000) 177–213.
[17] G. Chiola, C. Dutheillet, G. Franceschinis, S. Haddad, Stochastic well-formed coloured nets for symmetric modelling applications, IEEE Trans. Comput. 42 (1993) 1343–1360.
[18] G. Chiola, C. Dutheillet, G. Franceschinis, S. Haddad, A symbolic reachability graph for coloured Petri nets, Theor. Comput. Sci. 176 (1997) 39–65, https://doi.org/10.1016/S0304-3975(96)00010-2.
[19] M. Clavel, F. Durán, S. Eker, P. Lincoln, N.M. Oliet, J. Meseguer, C. Talcott, All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic, Lecture Notes in Computer Science, Springer, 2007.
[20] J. Dedeić, J. Pantović, J.A. Pérez, On primitives for compensation handling as adaptable processes, J. Log. Algebraic Methods Program. 121 (2021) 100675, https://doi.org/10.1016/j.jlamp.2021.100675, https://www.sciencedirect.com/science/article/pii/S2352220821000389.
[21] H. Ehrig, K. Hoffmann, J. Padberg, U. Prange, C. Ermel, Independence of net transformations and token firing in reconfigurable place/transition systems, in: J. Kleijn, A. Yakovlev (Eds.), Petri Nets and Other Models of Concurrency – ICATPN 2007, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 104–123.
[22] J. Esparza, R. Ledesma-Garza, R. Majumdar, P. Meyer, F. Niksic, An SMT-based approach to coverability analysis, in: A. Biere, R. Bloem (Eds.), Computer Aided Verification, Springer International Publishing, Cham, 2014, pp. 603–619.
[23] J. Esparza, M. Raskin, C. Weil-Kennedy, Parameterized analysis of immediate observation Petri nets, arXiv:1902.03025, 2019.
[24] H. Garavel, F. Lang, Equivalence Checking 40 Years After: A Review of Bisimulation Tools, Springer Nature Switzerland, Cham, 2022, pp. 213–265.
[25] C.A.R. Hoare, et al., Communicating Sequential Processes, vol. 178, Prentice-Hall, Englewood Cliffs, 1985.
[26] K. Hoffmann, H. Ehrig, T. Mossakowski, High-level nets with nets and rules as tokens, in: Proceedings of the 26th International Conference on Applications and Theory of Petri Nets, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 268–288.
[27] K. Jensen, Condensed state spaces for symmetrical coloured Petri nets, Form. Methods Syst. Des. 9 (1996) 7–40, https://doi.org/10.1007/BF00625967.
[28] T. Junttila, P. Kaski, Conflict propagation and component recursion for canonical labeling, in: International Conference on Theory and Practice of Algorithms in (Computer) Systems, Springer, 2011, pp. 151–162.
[29] T.A. Junttila, New canonical representative marking algorithms for place/transition-nets, in: J. Cortadella, W. Reisig (Eds.), Applications and Theory of Petri Nets 2004, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 258–277.
[30] L. Kahloul, A. Chaoui, K. Djouani, Modeling and analysis of reconfigurable systems using flexible Petri nets, in: F. Zavoral, J. Yaghob, P. Pichappan, E. El-Qawasmeh (Eds.), Networked Digital Technologies, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 343–357.
[31] M. Köhler-Bußmeier, Hornets: nets within nets combined with net algebra, in: G. Franceschinis, K. Wolf (Eds.), Applications and Theory of Petri Nets, Springer Berlin Heidelberg, 2009, pp. 243–262.
[32] M. Köhler-Bußmeier, On the complexity of the reachability problem for safe, elementary Hornets, Fundam. Inform. 129 (2014) 101–116. Dedicated to the memory of Professor Manfred Kudlek.
[33] M. Köhler-Bußmeier, A survey on decidability results for elementary object systems, Fundam. Inform. 130 (2014) 99–123.
[34] M. Köhler-Bußmeier, Restricting Hornets to support adaptive systems, in: W. van der Aalst, E. Best (Eds.), PETRI NETS 2017, Springer-Verlag, 2017.
[35] M. Köhler-Bußmeier, F. Heitmann, Complexity results for elementary Hornets, in: J.M. Colom, J. Desel (Eds.), PETRI NETS 2013, Springer-Verlag, 2013, pp. 150–169.
[36] O. Kummer, F. Wienberg, M. Duvigneau, J. Schumacher, M. Köhler, D. Moldt, H. Rölke, R. Valk, An extensible editor and simulation engine for Petri nets: renew, in: J. Cortadella, W. Reisig (Eds.), International Conference on Application and Theory of Petri Nets 2004, Springer-Verlag, 2004, pp. 484–493.
[37] M. Llorens, J. Oliver, Structural and dynamic changes in concurrent systems: reconfigurable Petri nets, IEEE Trans. Comput. 53 (2004) 1147–1158, https://doi.org/10.1109/TC.2004.66.
[38] A. Lluch Lafuente, J. Meseguer, A. Vandin, State space c-reductions of concurrent systems in rewriting logic, in: T. Aoki, K. Taguchi (Eds.), Formal Methods and Software Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 430–446.
[39] B.D. McKay, A. Piperno, Practical graph isomorphism, II, J. Symb. Comput. 60 (2014) 94–112, https://doi.org/10.1016/j.jsc.2013.09.003, https://www.sciencedirect.com/science/article/pii/S0747717113001193.
[40] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theor. Comput. Sci. 96 (1992) 73–155, https://doi.org/10.1016/0304-3975(92)90182-F.
[41] J. Meseguer, Membership algebra as a logical framework for equational specification, in: F.P. Presicce (Ed.), Recent Trends in Algebraic Development Techniques, Springer-Verlag, Berlin, Heidelberg, 1998, pp. 18–61.
[42] R. Milner, Communication and Concurrency, vol. 84, Prentice Hall, Englewood Cliffs, 1989.
[43] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, I, Inf. Comput. 100 (1992) 1–40.
[44] D. Neuen, P. Schweitzer, Benchmark graphs for practical graph isomorphism, CoRR, arXiv:1705.03686, 2017.

[45] J. Padberg, L. Kahloul, Overview of reconfigurable Petri nets, in: R. Heckel, G. Taentzer (Eds.), Graph Transformation, Specifications, and Nets: In Memory of Hartmut Ehrig, Springer International Publishing, Cham, 2018, pp. 201–222.
[46] J. Padberg, A. Schulz, Model checking reconfigurable Petri nets with maude, in: R. Echahed, M. Minas (Eds.), Graph Transformation, Springer International Publishing, Cham, 2016, pp. 54–70.
[47] U. Prange, H. Ehrig, K. Hoffmann, J. Padberg, Transformations in reconfigurable place/transition systems, in: P. Degano, R. De Nicola, J. Meseguer (Eds.), Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 96–113.
[48] W. Reisig, Petri Nets: An Introduction, Springer-Verlag New York, Inc., New York, NY, USA, 1985.
[49] K. Schmidt, Integrating low level symmetries into reachability analysis, in: S. Graf, M. Schwartzbach (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 315–330.
[50] P.H. Starke, Reachability analysis of Petri nets using symmetries, Syst. Anal. Model. Simul. 8 (1991) 293–303.
[51] M.O. Stehr, J. Meseguer, P.C. Ölveczky, Rewriting Logic as a Unifying Framework for Petri Nets, Springer-Verlag, Berlin, Heidelberg, 2001, pp. 250–303.
[52] R. Valk, Object Petri nets, in: J. Desel, W. Reisig, G. Rozenberg (Eds.), Lectures on Concurrency and Petri Nets: Advances in Petri Nets, Springer-Verlag, Berlin, Heidelberg, 2004, pp. 819–848.
[53] A. Valmari, Stubborn sets for reduced state space generation, in: Applications and Theory of Petri Nets, vol. 483, 1989, pp. 491–515.
[54] E. Viola, E-unifiability via narrowing, in: Theoretical Computer Science, Springer-Verlag, Berlin, Heidelberg, 2001, pp. 426–438.
[55] W.J. Yeh, M. Young, Compositional reachability analysis using process algebra, in: Proceedings of the Symposium on Testing, Analysis, and Verification, Association for Computing Machinery, New York, NY, USA, 1991, pp. 49–59.