Efficient contextual unfolding*

César Rodríguez[1], Stefan Schwoon[1], and Paolo Baldan[2]

[1] LSV, ENS Cachan & CNRS, INRIA Saclay, France
[2] Dipartimento di Matematica Pura e Applicata, Università di Padova, Italy

Abstract. A contextual net is a Petri net extended with read arcs, which allow transitions to check for tokens without consuming them. Contextual nets allow for better modelling of concurrent read access than Petri nets, and their unfoldings can be exponentially more compact than those of a corresponding Petri net. A constructive but abstract procedure for generating those unfoldings was proposed in earlier work; however, no concrete implementation existed. Here, we close this gap providing two concrete methods for computing contextual unfoldings, with a view to efficiency. We report on experiments carried out on a number of benchmarks. These show that not only are contextual unfoldings more compact than Petri net unfoldings, but they can be computed with the same or better efficiency, in particular with respect to the place-replication encoding of contextual nets into Petri nets.

1 Introduction

Petri nets are a means for reasoning about concurrent, distributed systems. They explicitly express notions such as concurrency, causality, and independence.

The unfolding of a Petri net is, essentially, an acyclic version of the net in which loops have been unrolled. The unfolding is infinite in general, but for finite-state Petri nets one can construct a finite complete prefix of it that completely represents the behaviour of the system, and whose acyclic structure permits easier analyses. This prefix is typically much smaller than the number of reachable markings because an unfolding exploits the inherently concurrent nature of the underlying system; loosely speaking, the more concurrency there is in the net, the more advantages unfoldings have over reachability-graph techniques.

Petri net unfoldings may serve as a basis for further analyses. There is a large body of work describing their construction, their properties, and their use in various fields (see, e.g., [6] for an extensive survey).

However, Petri nets are not well-suited to model concurrent read access, that is, multiple actions requiring non-exclusive access to one common resource. Consequently, the unfolding technique becomes inefficient in such situations. It is possible to mitigate this problem with a place-replication (PR) encoding [17]. Here, a resource with n readers is duplicated n times, and each reader obtains a "private" copy. However, the resulting unfolding may still be exponential in n.

* Supported by Fundación Caja Madrid, the MIUR project `SisteR`, and the University of Padua project `AVIAMO`.

Contextual nets explicitly model concurrent read accesses and address this problem. They extend Petri nets with *read arcs*, allowing an action to check for the presence of a resource without consuming it. They have been used, e.g., to model concurrent database access [13], concurrent constraint programs [12], priorities [9], and asynchronous circuits [17]. Their accurate representation of concurrency makes contextual unfoldings up to exponentially smaller in the presence of multiple readers, which promises to yield more efficient analysis procedures.

While the properties and construction of ordinary Petri net unfoldings are well-understood, research on how to construct and exploit the properties of contextual unfoldings has been lacking so far. Contextual unfoldings are introduced in [17, 1], and a first unfolding procedure for a restricted subclass can be found in [17]. A general but non-constructive procedure is proposed in [18].

A constructive, general solution was finally given in [3], at the price of making the underlying theory notably more complicated. In particular, computing a complete prefix required to annotate every event e with a subset of its histories, where roughly speaking, a history of e is a set of events that must precede e in any execution. However, it remained unclear whether the approach could be implemented with reasonable efficiency, and how. For 1-safe nets, the interest of computing a complete contextual prefix was doubtful: while the prefix can be exponentially smaller than the complete prefix of the corresponding PR-encoding, the intermediate product used to produce it has asymptotically the same size. More precisely, the number of histories in the contextual prefix matches the number of events in the PR-prefix (for general k-safe nets, this is not the case).

In [2], first theoretical advances towards an efficient implementation were made, proposing to annotate not only events, but conditions with histories. This gave rise to a binary concurrency relation, a concept that mimics a crucial element of efficient Petri unfolding tools [16, 10]. However, an implementation was still lacking, so the above doubts persisted.

In this paper, we address these open issues with the following contributions:

- We provide new approaches to two key elements of an unfolding tool: the computation of possible extensions and maintaining a concurrency relation.
- We generalise the results in [3, 2] in order to deal with a slight generalization of the adequate orders from [7]. Although not very surprising, this extension is quite relevant in practice as it drastically reduces the resulting prefixes.
- We implemented both approaches, aiming for efficiency. The resulting tool, called Cunf [14], matches dedicated Petri net unfolders like Mole [16] on pure Petri nets and additionally handles contextual unfoldings. The development of such a tool was non-trivial: First, the new unfolder is not a simple extension of an existing one because the presence of histories influences the data structures at every level. Secondly, even a Petri unfolder has complicated data structures, and its computation requires to solve subproblems that are computationally hard in principle [8].
- We ran the tool on a set of benchmarks and report on the experiments, for both approaches. In particular, it turns out that, even for 1-safe nets, our construction of contextual unfoldings is faster than that for PR-unfoldings.

Apart from details of the prefix computation, our main message is that efficient contextual unfolding is possible and performs better than the PR-encoding, even for 1-safe nets. Contextual nets and their unfoldings therefore have a rightful place in research on concurrency, including from an efficiency point of view. A full version of this paper including all the proofs can be found at [15].

2 Basic notions

A *contextual net (c-net)* is a tuple $N = \langle P, T, F, C, m_0 \rangle$, where P and T are disjoint sets of *places* and *transitions*, $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*, and $C \subseteq P \times T$ is the *context relation*. A pair $(p, t) \in C$ is called *read arc*. Any function $m \colon P \to \mathbb{N}$ is called a *marking*, and m_0 is the *initial marking*. A *Petri net* is a c-net without any read arcs.

For $x \in P \cup T$, we call ${}^\bullet x := \{y \in P \cup T \mid (y, x) \in F\}$ the *preset* of x and $x^\bullet := \{y \in P \cup T \mid (x, y) \in F\}$ the *postset* of x. The *context* of a place p is defined as $\underline{p} := \{\, t \in T \mid (p, t) \in C \,\}$, and the context of a transition t as $\underline{t} := \{\, p \in P \mid (p, t) \in C \,\}$. These notions are extended to sets in the usual fashion.

A marking m is *n-safe* if $m(p) \leq n$ for all $p \in P$. A set $A \subseteq T$ of transitions is *enabled* at m if for all $p \in P$,

$$m(p) \geq |p^\bullet \cap A| + \begin{cases} 1 & \text{if } \underline{p} \cap A \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

Such A can *occur* or *be executed*, leading to a new marking m', where $m'(p) = m(p) - |p^\bullet \cap A| + |{}^\bullet p \cap A|$ for all $p \in P$. We call $\langle m, A, m' \rangle$ a *step* of N.

A finite sequence of transitions $\sigma = t_1 \dots t_n \in T^*$ is a *run* if there exist markings $m_1, \dots, m_n$ such that $\langle m_{i-1}, \{t_i\}, m_i \rangle$ is a step for $1 \leq i \leq n$, and m_0 is the initial marking of N; if such a run exists, m_n is said to be *reachable*. A c-net N is said to be n-safe if every reachable marking of N is n-safe.

Fig. 1 (a) depicts a 1-safe c-net. Read arcs are drawn as undirected lines. For t_2, we have $\{p_1\} = {}^\bullet t_2$, $\{p_3\} = \underline{t_2}$ and $\{p_4\} = t_2^\bullet$.

General assumptions. We restrict our interest to finite 1-safe c-nets and treat markings as sets of places. Furthermore, for any c-net $N = \langle P, T, C, F, m_0 \rangle$ we assume for all transitions $t \in T$ that ${}^\bullet t \cap \underline{t} = \emptyset$; notice that transitions violating this condition can never fire in 1-safe nets.

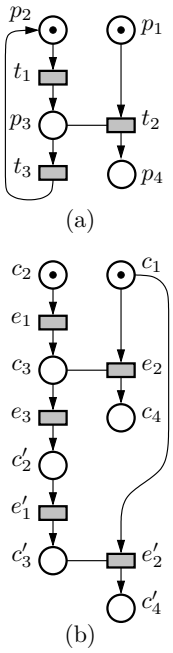

Fig. 1. (a) A 1-safe c-net; and (b) an unfolding prefix.

2.1 Encodings of contextual nets

A c-net N can be encoded into a Petri net whose reachable markings are in one-to-one correspondence with those of N. We treat two such encodings, and

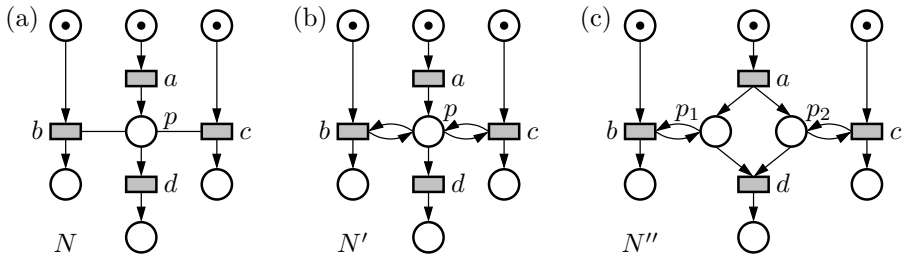

Fig. 2. C-net N, its *plain encoding* N' and its *Place-Replication encoding* N''.

illustrate them by the c-net N in Fig. 2 (a). Place p has two transitions b, c in its context, modelling a situation where, e.g., two processes are read-accessing a common resource modelled by p. Note that step $\{b, c\}$ can occur in N.

Plain encoding. Given a c-net N, the *plain encoding* of N is the net N' obtained by replacing every read arc (p, t) in the context relation by a read/write loop $(p, t), (t, p)$ in the flow relation. The net N' has the same reachable markings as N; it also has the same runs but not the same steps as N. An example can be found in Fig. 2 (b). Note that the step $\{b, c\}$ can no longer occur in N', as the firings of $\{b\}$ and $\{c\}$ are sequentialized.

PR-encoding. The *place-replication (PR-) encoding* [17] of a c-net N is a Petri net N'' in which we substitute every place p read by $n \geq 1$ transitions $t_1, \ldots, t_n$ by places $p_1, \ldots, p_n$, updating the flow relation of N'' as follows. For $i \in \{1, \ldots, n\}$,

1. transition t_i consumes and produces place p_i, i.e., $p_i \in {}^\bullet t_i$ and $p_i \in t_i^\bullet$;
2. any transition t producing p in N produces p_i in N'', i.e., $p_i \in t^\bullet$;
3. any transition t consuming p in N consumes p_i in N'', i.e., $p_i \in {}^\bullet t$.

A PR-encoding is depicted in Fig. 2 (c). Reachable markings, runs, and steps of N'' are in one-to-one correspondence to those of N.

3 Contextual unfoldings and their prefixes

In this section, we mostly recall basic definitions from [3] concerning unfoldings. We fix a 1-safe c-net $N = \langle P, T, F, C, m_0 \rangle$ for the rest of the section. Intuitively, the unfolding of N is a safe acyclic c-net where loops of N are "unrolled"; in general, this structure is infinite.

Definition 1. *The unfolding of N, written $\mathcal{U}_N$, is a c-net $(B, E, G, D, \widehat{m}_0)$ equipped with a mapping $f\colon (B \cup E) \to (P \cup T)$, which we extend to sets and sequences in the usual way. We call the elements of B* conditions*, and those of E* events*; f maps conditions to places and events to transitions.*

Conditions will take the form $\langle p, e' \rangle$, where $p \in P$ and $e' \in E \cup \{\bot\}$, and events will take the form $\langle t, M \rangle$, where $t \in T$ and $M \subseteq B$. We shall assume

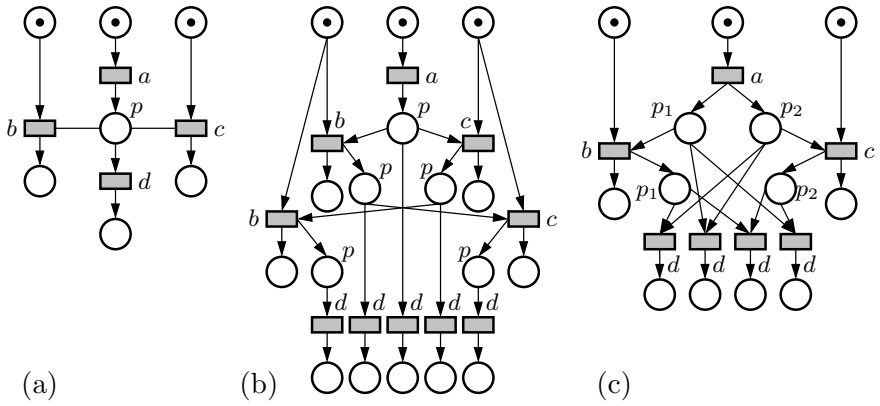

Fig. 3. Unfoldings of N, N', and N'' from Fig. 2

$f(\langle p, e' \rangle) = p$ and $f(\langle t, M \rangle) = t$, respectively. A set M of conditions is called concurrent*, written $conc(M)$, iff $\mathcal{U}_N$ has a reachable marking M' s.t. $M' \supseteq M$.*

$\mathcal{U}_N$ is the smallest net containing the following elements:

- *if $p \in m_0$, then $\langle p, \bot \rangle \in B$ and $\langle p, \bot \rangle \in \widehat{m}_0$;*
- *for any $t \in T$ and disjoint pair of sets $M_1, M_2 \subseteq B$ such that $conc(M_1 \cup M_2)$, $f(M_1) = {}^\bullet t$, $f(M_2) = \underline{t}$, we have $e := \langle t, M_1 \cup M_2 \rangle \in E$, and for all $p \in t^\bullet$, we have $\langle p, e \rangle \in B$. Moreover, G and D are such that ${}^\bullet e = M_1$, $\underline{e} = M_2$, and $e^\bullet = \{ \langle p, e \rangle \mid p \in t^\bullet \}$.*

Fig. 3 shows unfoldings of the nets from Fig. 2, where f is indicated by the labels of conditions and events. In this case, the c-net is isomorphic to its unfolding; crucially, it is smaller than the unfoldings of its two encodings. Call events labelled by b and c "readers", and events labelled by d "consumers". If, in Fig. 2, we replaced b, c by n transitions reading from p, there would be n readers and one consumer in the contextual unfolding; $\mathcal{O}(n!)$ readers *and* consumers in the plain unfolding; and n readers but 2^n consumers in the PR-unfolding.

$\mathcal{U}_N$ represents all possible behaviours of N, and, in particular m is reachable in N iff some $\widehat{m}$ with $f(\widehat{m}) = m$ is reachable in $\mathcal{U}_N$. Intuitively, the plain unfolding explodes because it represents the step $\{b, c\}$ of the c-net by two runs; and the cycles in the PR-encoding mean more consuming events for the PR-unfolding.

Definition 2. *The* causality relation *on $\mathcal{U}_N$, denoted $<$, is the transitive closure of $G \cup \{ (e, e') \in E \times E \mid e^\bullet \cap \underline{e'} \neq \emptyset \}$. For $x \in B \cup E$, we write $[x]$ for the set of* causes *of x, defined as $\{ e \in E \mid e \leq x \}$, where $\leq$ is the reflexive closure of $<$.*

In Fig. 1 (b), we have, e.g., $c_2 < e_1$, $e_1 < e_2$, and $c_2 < e_2$. The causality relation between a pair of events $e < e'$ captures the intuition that e must occur before e' in any run that fires e'.

Definition 3. *A set $X \subseteq E$ is called* causally closed *if $[e] \subseteq X$ for all $e \in X$. A* prefix *of $\mathcal{U}_N$ is a net $\mathcal{P} = \langle B', E', G', D', \widehat{m}_0 \rangle$ such that $E' \subseteq E$ is causally closed, $B' = \widehat{m}_0 \cup (E')^\bullet$, and G', D' are the restrictions of G, D to $(B' \cup E')$.*

In other words, a prefix is a causally-closed subnet of $\mathcal{U}_N$. Surely, if $\mathcal{P}$ is a prefix and $\widehat{m}$ a marking reachable in it, then $f(\widehat{m})$ is reachable in N. We are interested in computing a prefix for which the inverse also holds.

Definition 4. *A prefix $\mathcal{P}$ is called* complete *if for all markings m, m is reachable in N iff there exists a marking $\widehat{m}$ reachable in $\mathcal{P}$ such that $f(\widehat{m}) = m$.*

A complete prefix thus preserves all behavioural information about N, while being typically smaller than its reachability graph; yet its acyclic structure makes the reachability problem easier than for N itself [11]. Moreover, as we saw in Fig. 3, a contextual unfolding is more succinct than its corresponding Petri net unfolding. Other papers, e.g., [17], consider a slightly stronger notion of completeness imposing that not only reachable markings, but also firable transitions have a representative in the prefix. That would not affect the results in this paper.

4 Constructing finite prefixes

In this section, we make inroads on how to construct a finite prefix. The material from this section mostly recalls elements from [3], with minor modifications. We fix a net N and its unfolding $\mathcal{U}_N$ as in Section 3.

Consider events e_2 and e_3 in Fig. 1 (b). Clearly, $e_2 < e_3$ does not hold. However, any run that fires *both* e_2 and e_3 will fire e_2 before e_3 (since e_3 consumes c_3). This situation arises due to read arcs and motivates the next definition.

Definition 5. *Two events $e, e' \in E$ are in* asymmetric conflict*, written $e \nearrow e'$, iff (i) $e < e'$, or (ii) $\underline{e} \cap {}^\bullet e' \neq \emptyset$, or (iii) $e \neq e'$ and ${}^\bullet e \cap {}^\bullet e' \neq \emptyset$. For a set of events $X \subseteq E$, we write $\nearrow_X$ to denote the relation $\nearrow \cap (X \times X)$.*

Asymmetric conflict can be thought of as a scheduling constraint: if both e, e' occur in a run, then e must occur first. Note that in case (iii) this is vacuously the case, as e, e' cannot both occur. Thus, by condition (iii) $\nearrow$ subsumes the symmetric conflicts known from Petri net unfoldings as loops of length two.

Definition 6. *A* configuration *of $\mathcal{U}_N$ is a finite, causally closed set of events $\mathcal{C}$ such that $\nearrow_{\mathcal{C}}$ is acyclic. $Conf(\mathcal{U}_N)$ denotes the set of all such configurations.*

A set of events is a configuration iff all its events can be ordered to form a run that respects the scheduling constraints given by $\nearrow$. We say that configuration $\mathcal{C}$ *evolves* to configuration $\mathcal{C}'$, written $\mathcal{C} \sqsubseteq \mathcal{C}'$, iff $\mathcal{C} \subseteq \mathcal{C}'$ and $\neg(e' \nearrow e)$ for all $e \in \mathcal{C}$ and $e' \in \mathcal{C}' \setminus \mathcal{C}$. Intuitively, a run of $\mathcal{C}$ can be extended into a run of $\mathcal{C}'$.

Configurations $\mathcal{C}$, $\mathcal{C}'$ are said to be in *conflict*, written $\mathcal{C} \# \mathcal{C}'$, when there is no configuration $\mathcal{C}''$ verifying $\mathcal{C} \sqsubseteq \mathcal{C}''$ and $\mathcal{C}' \sqsubseteq \mathcal{C}''$. Note that if two configurations are *not* in conflict, then their union is a configuration.

The *cut* of a configuration $\mathcal{C}$ is the marking reached in $\mathcal{U}_N$ by a run of $\mathcal{C}$. We define $\mathsf{Cut}(\mathcal{C}) := (\widehat{m}_0 \cup \mathcal{C}^\bullet) \setminus {}^\bullet\mathcal{C}$. The *marking* of $\mathcal{C}$ is its image through f: $\mathsf{Mark}(\mathcal{C}) := f(\mathsf{Cut}(\mathcal{C}))$.

Definition 7. *Let e be an event. If $\mathcal{C}$ is a configuration with $e \in \mathcal{C}$, we define the configuration $\mathcal{C}[\![e]\!] := \{ e' \in \mathcal{C} \mid e'(\nearrow_{\mathcal{C}})^* e \}$ as the* history of e in $\mathcal{C}$. *Moreover, $Hist(e) := \{ \mathcal{C}[\![e]\!] \mid \mathcal{C} \in Conf(\mathcal{U}_N) \wedge e \in \mathcal{C} \}$ is the set of histories of e.*

While in Petri net unfoldings each event has exactly one history, a contextual unfolding may have multiple (even infinitely many) histories per event. For instance, in Fig. 1 (b) $Hist(e_3) = \{\{e_1, e_3\},\ \{e_1, e_2, e_3\}\}$. To compute a complete prefix, one annotates events with a finite subset of their histories.

Definition 8. *An* enriched event *is a pair $\langle e, H \rangle$ where $e \in E$ and $H \in Hist(e)$. A* closed enriched prefix (CEP) *of $\mathcal{U}_N$ is a pair $\mathcal{E} = \langle \mathcal{P}, \chi \rangle$ such that $\mathcal{P} = \langle B', E', G', D', \widehat{m}_0 \rangle$ is a prefix and $\chi \colon E' \to 2^{2^E}$ satisfies for all $e \in E'$ (i) $\emptyset \neq \chi(e) \subseteq Hist(e)$, and (ii) $H \in \chi(e)$ and $e' \in H$ imply $H[\![e']\!] \in \chi(e')$. For an enriched event $\langle e, H \rangle$, we write $\langle e, H \rangle \in \mathcal{E}$ if $e \in E'$ and $H \in \chi(e)$.*

In [3], a complete prefix of $\mathcal{U}_N$ is constructed by a saturation procedure that adds one enriched event at a time until there remains no addition that would "contribute" new markings. We concretize this idea in the following:

Definition 9. *Let $\mathcal{E}$ be a CEP. An enriched event $\langle e, H \rangle$ is a* possible extension *of $\mathcal{E}$ iff $\langle e', H[\![e']\!] \rangle \in \mathcal{E}$ for all $e' \in H$, $e' \neq e$, but $\langle e, H \rangle \notin \mathcal{E}$.*

Let $\prec$ be a partial order among configurations verifying that $\mathcal{C} \sqsubseteq \mathcal{C}'$ and $\mathcal{C} \neq \mathcal{C}'$ implies $\mathcal{C} \prec \mathcal{C}'$. We extend $\prec$ to enriched events by $\langle e, H \rangle \prec \langle e', H' \rangle$ if $H \prec H'$. Given a fixed $\prec$, a tuple $\langle e, H \rangle$ is called *cutoff* iff there exists an enriched event $\langle e', H' \rangle$ such that $\mathsf{Mark}(H') = \mathsf{Mark}(H)$ and $\langle e', H' \rangle \prec \langle e, H \rangle$. Thus, $\prec$ parametrizes the following informal algorithm:

Algorithm 1.

- Start with the CEP that contains just $\widehat{m}_0$;
- Then, in each iteration, add a non-cutoff $\prec$-minimal possible extension.
- If no non-cutoff possible extensions remain, terminate.

Whether Algorithm 1 terminates with a *complete* prefix depends on the choice of $\prec$. It was shown in [3, 2] that the procedure above yields a complete prefix if $\prec$ is the partial order due to McMillan [11]. However, it is known for Petri net unfoldings that using a total, so-called *adequate* order as defined in [7] can result in up to exponentially smaller complete prefixes.

Proposition 1. *Let N be a 1-safe c-net. If $\prec$ is adequate, then Algorithm 1 terminates with a CEP $\mathcal{E} = \langle \mathcal{P}, \chi \rangle$ such that $\mathcal{P}$ is a complete prefix of $\mathcal{U}_N$.*

5 Two approaches to possible extensions and concurrency

We now turn to the question of how to implement Algorithm 1 efficiently, for constructing unfoldings in practice. The main computational problem is to identify the possible extensions at each iteration of the procedure. Let N and $\mathcal{U}_N$ be as in the previous sections.

For Petri net unfolders (which do not deal with histories) this involves identifying sets M of conditions such that $conc(M)$ and $f(M) = {}^\bullet t$ for some $t \in T$ (compare Definition 1). For Petri nets, it is known that $conc(M)$ holds iff $conc(\{c_1, c_2\})$ for all pairs $c_1, c_2 \in M$. Possible extensions can therefore be identified by repeatedly consulting a *binary* relation on conditions. Moreover, this binary relation can be computed efficiently and incrementally during prefix construction. This idea is exploited by existing tools such as Mole [16] or Punf [10].

The above statement about $conc(\cdot)$ was shown to be invalid for contextual unfoldings in [3]. However, one can define a binary relation with similar properties on conditions enriched with histories.

Definition 10. *Let c be a condition. A* generating history *of c is $\emptyset$ if $c \in \widehat{m}_0$, or $H \in Hist(e)$, where $\{e\} = {}^\bullet c$. A* reading history *of c is any $H \in Hist(e)$ such that $e \in \underline{c}$. A* history *of c is any of its generating or reading histories or $H_1 \cup H_2$, where H_1 and H_2 are histories of c verifying $\neg(H_1 \# H_2)$. In the latter case, the history is called* compound.

If H is a history of c, we call $\langle c, H\rangle$ an *enriched condition*, called generating, reading, or compound condition, according to H[3]. For a CEP $\mathcal{E} = \langle \mathcal{P}, \chi\rangle$, we say $\langle c, H\rangle \in \mathcal{E}$ if H is built from histories in χ. The mapping f is extended to enriched events and conditions by $f(\langle e, H\rangle) = f(e)$ and $f(\langle c, H\rangle) = f(c)$.

Definition 11. *Two enriched conditions $\langle c, H\rangle, \langle c', H'\rangle$ are called* concurrent*, written $\langle c, H\rangle \parallel \langle c', H'\rangle$, iff $\neg(H \# H')$ and $c, c' \in \mathsf{Cut}(H \cup H')$.*

In Section 5.1, we discuss how $\parallel$ helps to compute possible extensions. In Section 5.2 we then discuss how to update $\parallel$ during the unfolding construction.

5.1 Computing possible extensions

We discuss two ways of computing possible extensions. The first, called "lazy", avoids constructing compound conditions (see Definition 10), reducing the number of enriched conditions considered. The second, "eager" approach does use compound conditions, saving work while computing possible extensions instead. The lazy approach was introduced in [2] for the McMillan order, but holds also for the total order of [7]. The eager approach is proposed for the first time here.

Lazy Approach. The lazy approach [2] is based on the observation that the history associated with an event can be constructed by taking generating and read histories for places in the pre-set and generating histories for places in the context. This is expressed by the following proposition:

Proposition 2. *[2] The pair $\langle e, H\rangle$ with $f(e) = t$ is an enriched event iff there exist sets X_p, X_c of enriched conditions such that*

[3] In [2], generating histories were called *causal*; we find the term generating more suggestive. The definition of compound histories is new and does not appear in [2].

1. $f(X_p) = {}^\bullet t$ *and* $f(X_c) = \underline{t}$*;*
2. $X_p \cup X_c$ *contains exactly one generating condition for every* $c \in ({}^\bullet e \cup \underline{e})$*;*
3. X_p *contains generating or reading conditions,* X_c *generating conditions;*
4. *for all* $\rho, \rho' \in X_p \cup X_c$ *we have* $\rho \parallel \rho'$*;*
5. *finally,* $H = \bigcup_{\langle c, H' \rangle \in X_p \cup X_c} H'$*.*

Proposition 2 allows to identify new possible extensions whenever a prefix is extended with new enriched conditions. Compound conditions are avoided at the price of combining generating and reading conditions as stated in items 2–4 for every possible extension.

Eager approach. The eager approach, instead of attempting to combine generating and reading histories when computing a possible extension, explicitly produces all types of enriched conditions, including compound ones. This means more enriched conditions, but on the other hand less work when computing possible extensions.

Proposition 3. *The pair* $\langle e, H \rangle$ *with* $f(e) = t$ *is an enriched event iff there exist sets* X_p, X_c *of enriched conditions such that*

1. $f(X_p) = {}^\bullet t$ *and* $f(X_c) = \underline{t}$*;*
2. $X_p \cup X_c$ *contains exactly one enriched condition for every* $c \in ({}^\bullet e \cup \underline{e})$*;*
3. X_p *contains arbitrary enriched conditions,* X_c *generating conditions;*
4. *for all* $\rho, \rho' \in X_p \cup X_c$ *we have* $\rho \parallel \rho'$*;*
5. *finally,* $H = \bigcup_{\langle c, H' \rangle \in X_p \cup X_c} H'$*.*

Notice that $|X_p| = |{}^\bullet t|$ in Proposition 3 whereas no such bound exists in Proposition 2. Like the latter, Proposition 3 allows to identify new possible extensions upon addition of new enriched conditions.

5.2 Updating the concurrency relation

We face the problem of keeping up to date the concurrency relation on enriched conditions when the unfolding grows by the insertion of new enriched events.

In [2] an approach is proposed, based on the introduction of another binary relation on enriched conditions, called subsumption. Intuitively, $\langle c, H \rangle$ *subsumes* $\langle c', H' \rangle$, written $\langle c, H \rangle \propto \langle c', H' \rangle$, when in the history H there is an event that reads condition c', with history H', and c' is not consumed by H. This means that when taking the enriched condition $\langle c, H \rangle$ we are also implicitly taking $\langle c', H' \rangle$. For instance, in Fig. 1(b), $\langle c_4, \{e_1, e_2\} \rangle$ subsumes $\langle c_3, \{e_1, e_2\} \rangle$. When a new enriched event is inserted in the unfolding, subsumption plays a role in updating the concurrency relation. Assume that the inserted event is $\langle e, H \rangle$ and that it is created using sets X_c, X_p (see Proposition 2 or Proposition 3). Then the enriched conditions generated by $\langle e, H \rangle$ are concurrent with an enriched condition ρ already in the prefix iff $X_p \cup X_c \cup \{\rho\}$ is pairwise concurrent and it satisfies suitable closure properties w.r.t subsumption.

Here we show that for 1-safe nets the result below holds, which allows to update the concurrency relation for a new generating or reading conditions inserted in the unfolding, in a simpler way, without the need of computing subsumption.

Proposition 4. *In Algorithm 1, let $\mathcal{E}$ be the current CEP, where $\langle e, H\rangle$ is the last addition thanks to sets X_c, X_p as per Proposition 2 or Proposition 3. We denote by $Y_p = e^\bullet \times \{H\}$ and $Y_c = \underline{e} \times \{H\}$ the generating and reading conditions created by the addition of $\langle e, H\rangle$. Let $\rho \in Y_p \cup Y_c$, and let $\rho' = \langle c', H'\rangle \in \mathcal{E}$ be any other enriched condition. Then $\rho \parallel \rho'$ iff*

$$\rho' \in Y_p \cup Y_c \ \vee \ (c' \notin {}^\bullet e \ \wedge \ \forall \rho_1 \in X_p \cup X_c : (\rho_1 \parallel \rho') \ \wedge \ {}^\bullet\underline{e} \cap H' \subseteq H)$$

Then the concurrency relation can be transferred to compound conditions on the basis of the result below.

Proposition 5. *Let $\rho = \langle c, H_1 \cup H_2\rangle$ be a compound condition of $\mathcal{E}$, where $\rho_1 = \langle c, H_1\rangle$, $\rho_2 = \langle c, H_2\rangle$ are enriched conditions verifying $\neg(H_1 \# H_2)$. Let $\rho' \in \mathcal{E}$ be any enriched condition. Then $\rho \parallel \rho'$ iff $\rho_1 \parallel \rho' \wedge \rho_2 \parallel \rho'$.*

5.3 Discussion: lazy vs. eager approach

In order to discover possible extensions of the form $\langle e, H\rangle$, both approaches consider combinations of generating and reading histories for conditions $c \in {}^\bullet e$.

Consider Proposition 2. For every possible extension, the lazy approach takes one generating and possibly multiple reading histories for c, all of which must be concurrent. If the events in $\underline{c}$ have many different histories, or $\underline{c}$ is large, then many different combinations need to be checked for concurrency.

The eager approach (Proposition 3) takes exactly one enriched condition of arbitrary type, including compound, for c. Compound histories are a set of concurrent reading histories (Definition 10); thus a compound condition represents pre-computed information needed to identify possible extensions.

We consider two examples where eager beats lazy and vice versa. In Fig. 4 (a), condition c has a sequence of n readers and hence $n+1$ histories $\{e_1, \ldots, e_i\}$, for $i = 0, \ldots, n$. For each history H of c', eager simply combines H with the $n+1$ histories for c, while lazy checks all 2^n subsets of $e_1, \ldots, e_n$ to find these $n+1$ compound histories. If c' has many histories, eager becomes largely superior. Of course, an intelligent strategy may help lazy to avoid exploring all 2^n subsets one by one. However, even with a good strategy, lazy still has to enumerate at least the same combinations as eager; and since the problem of identifying the useful subsets is NP-complete [8], there will always be instances where lazy becomes inefficient, whatever strategy is employed.

On the other hand, consider Fig. 4 (b). Again, c has n readers, this time yielding 2^n histories. Suppose that $f(c)$ is an input place of some transition t. Now, if t also has $f(a)$ and $f(b)$ in its preset, then no t-labelled event e will ever be generated in the unfolding, and all histories of c are effectively useless. Since those compound conditions also appear in the computation of the concurrency relation, they become a liability in terms of both memory and execution time. The lazy approach does not suffer from this problem here.

Both approaches therefore have their merits, and we implemented them both. We shall report on experiments in Section 7. Concerning Section 5.2, we only retained the new approach, which is clearly better than that of [2].

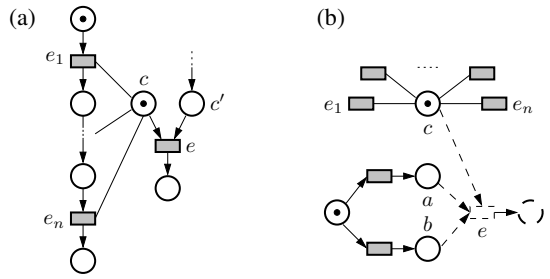

Fig. 4. Good examples for the eager (a) and the lazy (b) approach.

6 Efficient prefix construction

We implemented the procedure from Algorithm 1, using the methods proposed in Section 5. The resulting tool, called Cunf, is publicly available [14]. Cunf expects as input a 1-safe c-net and produces as output a complete unfolding prefix.

Notice that efficient tools exist for the unfolding of Petri nets such as Mole [16] or Punf [10]. While we profited much from the experiences gained from developing Mole, Cunf is not a simple extension of Mole. The issues of asymmetric conflict and histories permeate every aspect of the construction so that we went for a completely new implementation in C, comprising some 4,000 lines of code.

Here, we review some features such as data structures and implementation details, relevant to handling the complications imposed by contextual unfoldings, that helped to produce an efficient tool. Experiments are reported in Section 7.

The history graph. Cunf needs to maintain enriched events and conditions, i.e. tuples $\langle e, H\rangle$ or $\langle c, H\rangle$, where H is a history. We store these in a graph structure, maintained while the enriched prefix $\mathcal{E}$ evolves. Formally, the *history graph* associated with $\mathcal{E}$ is a directed graph $\mathcal{H}_\mathcal{E}$ whose nodes are the enriched events of $\mathcal{E}$, and with edges $\langle e, H\rangle \to \langle e', H'\rangle$ iff $e' \in H$ and $H' = H[\![e']\!]$ and either (i) $(e'^\bullet \cup \underline{e'}) \cap {}^\bullet e \neq \emptyset$ or (ii) $e'^\bullet \cap \underline{e} \neq \emptyset$. Each node $\langle e, H\rangle$ is labelled by e.

Intuitively, $\mathcal{H}_\mathcal{E}$ has an edge between two enriched events $\langle e, H\rangle$ and $\langle e', H'\rangle$ iff some enriched condition $\langle c, H'\rangle$ has been used to construct $\langle e, H\rangle$ (in the sense of Proposition 2 or Proposition 3).

This structure allows Cunf to perform many operations efficiently: every additional enriched event enlarges the graph by just one node plus some edges; common parts of histories are shared. We can easily enumerate the events in $H \in \chi(e)$ by following the edges from node $\langle e, H\rangle$, and $\mathcal{H}_\mathcal{E}$ implicitly represents the relation $\sqsubset$. Given an event e, we can enumerate the histories in $\chi(e)$ by keeping a list of nodes in $\mathcal{H}_\mathcal{E}$ labelled by e. Given a condition c, we can enumerate its generating and reading histories similarly.

Compound conditions are stored in a shared-tree-like structure, where leaves represent reading histories and internal nodes compound histories. An internal node has two children, one of which is a leaf, the other either internal or a leaf.

One easily sees that a compound history of c corresponds, w.l.o.g., to a union $H_1 \cup \cdots \cup H_n$ of reading histories. Every internal node represents such a union, and the structure allows sharing if one compound history contains another.

Possible extensions. Cunf behaves similar to Mole or other unfolders in its flow of logic, but its actions are on enriched events and conditions. We start with a prefix containing just $\widehat{m}_0$ and identify the initial possible extensions. As long as the set of possible extensions is non-empty, we choose a "minimal" extension and add it unless it is a cutoff. For "minimal", we use the adequate order $\prec_F$ from [7]. Adding $\langle e, H\rangle$ means adding H to $\chi(e)$, creating e first if necessary. The addition of $\langle e, H\rangle$ will give rise to various types of enriched conditions for whom we compute the concurrency relation (see below). Whenever we add an enriched condition ρ, we attempt to find possible extensions, i.e. sets X_p, X_c matching the conditions in Propositions 2 or 3 such that $X_p \cup X_c$ includes ρ, where, in order to implement condition 4, we use the precomputed binary concurrency relation. Upon identifying a possible extension $\langle e, H\rangle$, we immediately compute its marking, information relevant to deciding $\prec_F$, and certain lists $r(H), s(H)$ during two linear traversals of H. Details on $r(H)$ and $s(H)$ are given below.

Concurrency relation. The relation $\|$ on the enriched conditions $\mathcal{E}$ can be stored and updated whenever new possible extensions are appended to $\mathcal{E}$. We detail now how Propositions 4 and 5 are used to efficiently compute this update.

Let $c(\rho)$ denote the set of enriched conditions ρ' verifying $\rho \| \rho'$. The relation $\|$ is generally sparse, and Cunf stores $c(\rho)$ as a list. However, for the purpose of the following, $c(\rho)$ could also be a row in a matrix representing $\|$.

For reading and generating conditions ρ (Proposition 4), Cunf initially sets $c(\rho)$ to $Y_p \cup Y_c$. Next, it computes the intersection of $c(\rho')$ for all $\rho' \in X_p \cup X_c$, and filters out those $\langle c', H'\rangle$ for which $\underline{{}^\bullet e} \cap H' \not\subseteq H$ holds. In order to compute this condition without actually traversing H and H', we use the sets $r(H)$ and $s(H)$ computed earlier (see above). These are defined as $r(H) := \{\, e' \in H \mid \underline{e'} \cap \mathsf{Cut}(H) \neq \emptyset \,\}$ and $s(H) := \{\, e' \in H \mid e' \in \underline{{}^\bullet e} \,\}$. Then $\underline{{}^\bullet e} \cap H' \not\subseteq H$ holds iff $\underline{{}^\bullet e} \setminus s(H) \cap r(H') \neq \emptyset$, which can be computed traversing $\underline{{}^\bullet e}$ and $s(H)$ one time, and checking $r(H')$ for every ρ'. Note that, while the other steps have their counterparts in Petri net unfoldings, this step is new and specific to c-nets. However, we find that this implementation keeps the overhead very small.

As for compound conditions ρ built using ρ_1 and ρ_2 (Proposition 5), Cunf computes $c(\rho)$ as the intersection of $c(\rho_1)$ and $c(\rho_2)$.

Certain enriched conditions $\rho = \langle c, H\rangle$ need not to be included in the concurrency relation. It is safe, for instance, to leave $c(\rho)$ empty if ρ is generating and $f(c)^\bullet \cup \underline{f(c)} = \emptyset$, or if H is a cutoff. We can also avoid computing $c(\rho)$ if ρ is reading or compound and $f(c)^\bullet = \emptyset$, even if $\underline{f(c)} \neq \emptyset$.

7 Experiments

In order to experimentally evaluate our tool, we performed a series of experiments. We were interested in the following questions:

	Plain		PR		Contextual				Ratios		
Net	Events	t_P	Events	t_R	Av. $\underline{t}$	Events	t_L	t_E	t_E/t_P	t_E/t_R	t_E/t_L
bds_1.sync	12900	0.51	4302	0.26	1.22	1866	0.14	0.14	0.27	0.54	1.00
byzagr4_1b	14724	3.40	8044	5.30	0.92	8044	3.41	2.90	0.85	0.55	0.85
dpd_7.sync	10457	0.88	10457	0.99	0.77	10457	0.92	0.91	1.03	0.92	0.99
elevator_4	16856	2.01	16856	504.77	1.19	16856	1.27	1.26	0.63	>0.01	0.99
ftp_1.sync	83889	76.74	50928	113.38	1.05	50928	34.25	34.21	0.45	0.30	1.00
furnace_4	146606	40.39	100260	43.52	0.85	95335	23.48	18.34	0.45	0.42	0.78
key_4.fsa	67954	2.21	21742	4.30	0.37	4754	2036.66	6.33	2.86	1.47	>0.01
q_1.sync	10722	1.21	10722	2.18	0.90	10722	1.13	1.13	0.93	0.52	1.00
rw_12.sync	98361	3.95	98361	7.64	0.99	98361	4.52	3.10	0.78	0.41	0.69
rw_1w3r	15401	0.38	14982	0.69	0.48	14490	0.45	0.45	1.18	0.65	1.00
rw_2w1r	9241	0.30	9241	8.95	0.76	9241	0.43	0.40	1.33	0.04	0.93

Table 1. Experimental results

- Is the contextual unfolding procedure efficient?
- What is the size of the unfoldings, compared to Petri net unfoldings?
- How do the various approaches (lazy, eager, PR, plain encoding) compare?

Concerning the second and third point, contextual unfoldings may be up to exponentially more succinct than Petri net unfoldings, and we could contrive examples showing arbitrarily large discrepancies. To get more realistic numbers, we took a set of 1-safe nets provided in [4]. This collects nets with various characteristics that allowed to test practically all aspects of our implementation.

For each net N in the set, we first obtained the c-net N' by substituting pairs of arcs (p,t) and (t,p) in N by read arcs. Evidently, the plain encoding of N' is N. Secondly, we obtained the PR-encoding N'' of N'.

We first ran both Mole [16] and Cunf on the nets N and N'', which are ordinary Petri nets without read arcs. Naturally, both tools compute the same result; the object of this exercise was to establish whether Cunf was working reasonably efficient on known examples. Indeed, its running times were always within 70% and 140% of those of Mole, the differences due to minor implementation choices. To abstract from these details, we used Cunf for all further comparisons.

We then used Cunf to produce complete unfoldings of the plain net N, the PR-encoding N'', and of N' using both lazy and eager methods and the order $\prec_F$ from [7]. Table 1 summarizes the results. For all approaches, we list the number of events in the complete prefix and the running times (in seconds) of the eager approach. For c-nets, we additionally list the running time t_L of the lazy method, since only in proper c-nets this time differs from eager. Notice that the number of events in lazy and eager is the same; moreover, the number of *enriched* events in lazy and eager equals the number of events in PR (compare the discussion in the introduction). The average transition context size is provided for the c-nets, as well as three ratios comparing our running times.

We make the following observations:

– In all examples that we tried, the eager approach was always at least as fast as the lazy approach; an effect similar to the one in Fig. 4 (b) did not happen. On the other hand, in many examples both approaches were nearly equivalent, while in one case (`key_4`) lazy performed badly; see below.
– The eager approach handles all examples gracefully. It is significantly faster than the plain approach in half the cases, and significantly slower in only one case, `key_4`.
– The contextual methods produce smaller unfoldings than the plain approach in 6 out of 11 cases. Interestingly, these are not the same as those on which they run faster. For `elevator_4` and `rw_12.sync`, the same number of events is produced more quickly. Here, the read arcs are arranged in such a way that each event still has only one history; the time saving comes from the fact that the contextual approach produces fewer *conditions* and hence a smaller concurrency relation. For `key_4` and `rw_1w3r`, the contextual methods produce smaller unfoldings but take longer to run; see below for an explanation.
– Comparing with PR, the eager approach is consistently more efficient except for `key_4`. This clear tendency is slightly surprising given that the enriched contextual prefix has essentially the same size as the PR-prefix. We experimentally traced the difference to the enlarged presets of certain transitions in the PR-encoding (see Fig. 2), causing combinatorial overhead and increasing the number of conditions in the concurrency relation. Note that the ratio between number of events in contextual and number of events in PR is the average number of histories per event in the contextual approach.

We briefly discuss `key_4`, which causes problems for the contextual approaches. In this net, there is one place p with a read arc to almost every transition in the net, similar to Fig. 4 (a), with long sequences of readers. As discussed in Section 5.3, the eager approach constructs a number of enriched conditions linear in the length of each sequence whereas the lazy approach breaks down. The plain encoding works fast because every event creates a new copy of p, and every condition is concurrent with only one such copy. It remains to be seen whether the eager approach can be adapted to handle this special case in the same way.

8 Conclusions

We made theoretical and practical contributions to the computation of unfoldings of contextual nets. To our knowledge, Cunf is the first tool that efficiently produces these objects. The availability of a tool that produces contextual unfoldings may trigger new interest in applications of c-nets and the algorithmics of asymmetric event structures in general.

It will be interesting to explore the applications in verification. Unfolding-based techniques need two ingredients: an efficient method for generating them, and efficient methods for analyzing the prefixes. We have provided the first ingredient in this quest. We believe that traditional unfolding-based verification techniques [5] (e.g., SAT-based techniques) can be extended to work with contex-

tual unfoldings and that their succinctness may help to speed up these analyses. We find this topic to be an interesting avenue for future research.

Moreover, despite promising results, the present work will probably not be the last word on the algorithmics of contextual unfoldings; we have some ideas on how to further speed up the process. It would also be interesting to investigate a mix between eager and lazy that tries to get the best of the two worlds. For instance, one could start with the eager approach and switch (selectively for some conditions) to lazy as soon the number of compound conditions exceeds a certain bound. This, and other ideas, remain to be tested.

References

1. Baldan, P., Corradini, A., Montanari, U.: An event structure semantics for P/T contextual nets: Asymmetric event structures. In: Proc. of FoSSaCS '98. LNCS, vol. 1378, pp. 63–80. Springer (1998)
2. Baldan, P., Bruni, A., Corradini, A., König, B., Schwoon, S.: On the computation of McMillan's prefix for contextual nets and graph grammars. In: Proc. of ICGT'10. LNCS, vol. 6372, pp. 91–106 (2010)
3. Baldan, P., Corradini, A., König, B., Schwoon, S.: McMillan's complete prefix for contextual nets. ToPNoC 1, 199–220 (2008), LNCS 5100
4. Corbett, J.C.: Evaluating deadlock detection methods for concurrent software. IEEE Transactions on Software Engineering 22, 161–180 (1996)
5. Esparza, J., Heljanko, K.: Implementing LTL model checking with net unfoldings. In: Proc. of SPIN'01. LNCS, vol. 2057, pp. 37–56. Springer (2001)
6. Esparza, J., Heljanko, K.: Unfoldings - A Partial-Order Approach to Model Checking. EATCS Monographs in Theoretical Computer Science, Springer (2008)
7. Esparza, J., Römer, S., Vogler, W.: An improvement of McMillan's unfolding algorithm. Formal Methods in System Design 20, 285–310 (2002)
8. Heljanko, K.: Deadlock and Reachability Checking with Finite Complete Prefixes. Licentiate's thesis, Helsinki University of Technology (1999)
9. Janicki, R., Koutny, M.: Invariant semantics of nets with inhibitor arcs. In: Proc. of CONCUR'91. LNCS, vol. 527, pp. 317–331 (1991)
10. Khomenko, V.: Punf, `http://homepages.cs.ncl.ac.uk/victor.khomenko/tools/punf/`
11. McMillan, K.L.: Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In: Proc. of CAV'92. LNCS, vol. 663, pp. 164–177 (1992)
12. Montanari, U., Rossi, F.: Contextual occurrence nets and concurrent constraint programming. In: Dagstuhl Seminar 9301. LNCS, vol. 776 (1994)
13. Ristori, G.: Modelling Systems with Shared Resources via Petri Nets. Ph.D. thesis, Department of Computer Science, University of Pisa (1994)
14. Rodríguez, C.: Cunf, `http://www.lsv.ens-cachan.fr/~rodriguez/tools/cunf/`
15. Rodríguez, C., Schwoon, S., Baldan, P.: Efficient contextual unfolding. Tech. Rep. LSV-11-14, LSV, ENS de Cachan (2011)
16. Schwoon, S.: Mole, `http://www.lsv.ens-cachan.fr/~schwoon/tools/mole/`
17. Vogler, W., Semenov, A.L., Yakovlev, A.: Unfolding and finite prefix for nets with read arcs. In: Proc. of CONCUR'98. LNCS, vol. 1466, pp. 501–516 (1998)
18. Winkowski, J.: Reachability in contextual nets. Fundamenta Informaticae 51(1–2), 235–250 (2002)