Efficient Unfolding of Coloured Petri Nets Using Interval Decision Diagrams

Martin Schwarick[1], Christian Rohr[1], Fei Liu[2], George Assaf[1], Jacek Chodak[1], and Monika Heiner[1](✉)

[1] Brandenburg Technical University (BTU), Cottbus, Germany
monika.heiner@b-tu.de
[2] South China University of Technology, Guangzhou, China
https://www-dssz.informatik.tu-cottbus.de

Abstract. We consider coloured Petri nets, qualitative and quantitative ones alike, as supported by our PetriNuts tool family, comprising, among others, Snoopy, Marcie and Spike. Currently, most analysis and simulation techniques require to unfold the given coloured Petri net into its corresponding plain, uncoloured Petri net representation. This unfolding step is rather straightforward for finite discrete colour sets, but tends to be time-consuming due to the potentially huge number of possible transition bindings. We present an unfolding approach building on a special type of symbolic data structures, called Interval Decision Diagram, and compare its runtime performance with an unfolding engine employing an off-the-shelf library to solve constraint satisfaction problems. For this comparison we use the 22 scalable coloured models from the MCC benchmark suite, complemented by a few from our own collection.

Keywords: Coloured Petri nets · Unfolding · Symbolic data structures · Interval decision diagrams

1 Motivation

We consider coloured Petri nets, comprising coloured qualitative (i.e., time-free) Petri nets ($\mathcal{PN}^{\mathcal{C}}$) as well as coloured quantitative ones, consisting of coloured stochastic Petri nets ($\mathcal{SPN}^{\mathcal{C}}$), coloured continuous Petri nets ($\mathcal{CPN}^{\mathcal{C}}$) and coloured hybrid Petri nets ($\mathcal{HPN}^{\mathcal{C}}$), as supported by our PetriNuts tool family, including, among others, the modelling and simulation tool Snoopy [9], the analysis and simulation tool Marcie [10], and the simulation tool Spike [2].

Coloured Petri nets are a powerful modelling formalism, which has been used in all sorts of applications in various fields, covering natural sciences, engineering sciences and life sciences alike. But modelling is only one side of the coin. The other side calls for exploration of the model behaviour. Petri nets come generally along with a wide range of analysis techniques, extending from structural to behavioural analysis, and may include model animation to gain some initial trust in the model behaviour, as well as a variety of sophisticated simulation techniques in the case of stochastic, continuous or hybrid Petri nets.

R. Janicki et al. (Eds.): PETRI NETS 2020, LNCS 12152, pp. 324–344, 2020.
https://doi.org/10.1007/978-3-030-51831-8_16

However, most analysis and simulation techniques require currently an unfolding of the given coloured Petri net into its corresponding plain, uncoloured Petri net representation. That's also the case for our tool family. Thus, $\mathcal{PN}^{\mathcal{C}}$ are unfolded to $\mathcal{PN}$, $\mathcal{SPN}^{\mathcal{C}}$ to $\mathcal{SPN}$, $\mathcal{CPN}^{\mathcal{C}}$ to $\mathcal{CPN}$, and $\mathcal{HPN}^{\mathcal{C}}$ to $\mathcal{HPN}$; see [9] for the supported modelling paradigms and the relations among them.

When we confine ourselves to coloured Petri nets with finite discrete colour sets, this unfolding step is rather straightforward from an algorithmic point of view, but tends to be fairly time-consuming due to the potentially huge number of possible transition bindings. This situation may even get worse for quantitative Petri nets, because the transitions' rate functions can be colour-dependent, too.

Initially, we followed the unfolding approach proposed in [20], which gave us our first unfolding engine, which we called *Generic unfolding*. This engine applies templates and basically uses a similar pattern matching mechanism as CPN tools. But with increasing size of the generally scaleable coloured models, we observed an annoying increase of the unfolding runtime. Thus, we suggested in [22] an unfolding approach which deploys an off-the-shelf Constraint Satisfaction Problem (CSP) solver by means of the constraint solver library Gecode [6]. Its implementation, let's call it *Gecode unfolding*, brought some relief by a notable acceleration of the unfolding step.

However, there were always some doubts whether the Gecode-based CSP approach would be the ideal solution for the problem on hand. A CSP solver must generally find *one* solution for a given CSP. In contrast, unfolding of coloured Petri nets requires to iterate efficiently over the complete solution space.

Motivated by our long-term overwhelmingly positive experience with Interval Decision Diagrams (IDD) [11,12,30,34] and inspired by the idea of IDD-based creation of net transitions defined by guarded reward structures [32], we designed an IDD-based representation of the unfolding solution space. Currently, the IDD unfolding engine is integrated into Snoopy, Marcie and Spike. All three tools deal with coloured qualitative and/or quantitative Petri Nets according to [25] and communicate via files in CANDL (Coloured Abstract Net Description Language), a human-readable proprietary text format used within the PetriNuts tool family.

In this paper we explain step by step the general approach of IDD-based unfolding; it is organised as follows. We start off with recalling the basic principles of coloured Petri nets, followed by a gentle introduction into IDDs, where we illustrate IDD-based unfolding by an easy-to-follow simple example. For those interested in more algorithmic details, we describe next the implementation strategy in pseudocode notation. We evaluate our IDD unfolding engine by comparing its runtime with the one required by the Gecode unfolding engine. For this purpose, we consider coloured models representing different application scenarios, comprising all scalable (qualitative) coloured models from the PNML benchmark suite collected over the years for the Model Checking Contest (MCC) [15], complemented by quantitative coloured models from our own collection. We conclude the paper with a brief summary and outlook on future work.

We have deliberately written this paper in an informal way, avoiding any formal definitions and thus (self-) plagiarism; instead we provide pointers to literature, where the interested reader will find the formalisation of the concepts applied in this paper. This not only allows us to cope with the given page limit, but hopefully also contributes to the paper's readability.

2 Coloured Petri Nets

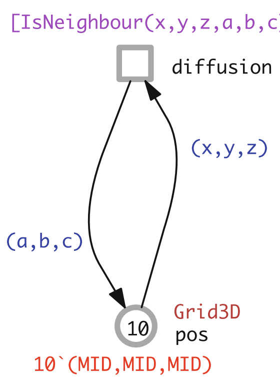
We briefly recall in an informal way some of the core ideas paving the way for the success story of coloured Petri nets; see [22] for more details and formal definitions of the coloured Petri nets as applied in this paper.

Coloured Petri nets [7,18] build on plain (uncoloured) Petri nets; thus they are equally made of places, transitions and arcs, follow the same basic principles of bipartite graphs, and enjoy an execution semantics, just as their uncoloured predecessors do. Additionally, coloured Petri nets are enriched with a tailored, rather sophisticated concept of (finite) discrete data types, called colour sets and inspired by high-level programming languages. Typical colour sets provided include integer subranges and enumeration sets. These colour sets induce the necessity for a couple of related net annotations, which in turn require some basic programming skills. Annotations often make use of guards, which are technically Boolean expressions over colour variables and constants, possibly involving user-defined functions.

- *Places* - get assigned colour sets and may contain a multiset of distinguishable tokens, each coloured with a colour of the place's colour set. The initial marking definition may involve guards to specify subsets of the place's colour set, where each colour of a given subset shall get the same amount of tokens.
- *Transitions* - get assigned guards, which must be evaluated to true for enabling the transition. The trivial guard *true* is usually not explicitly given.
- *Arcs* - get assigned colour expressions, over colour variables and constants, possibly involving user-defined functions – just like guards, but the result type of a colour expression is a multiset over the colour set of the connected place. This colour expression may also incorporate guards to specify different colour expressions for different conditions.
- *Rate functions* - Quantitative nets involve transition rate functions, which may be colour-dependent. This requires again guards to assign different rate functions for different colours.

These colour-related annotations permit, among others, to conveniently encode a regular grid where each grid position represents a specific point in space; see Fig. 1 for a related introductory example, encoding diffusion in 3D space as $\mathcal{CPN}^{\mathcal{C}}$. Here, colours serve as addresses for grid positions, or to put it differently: movement in space boils down to recolouring of tokens; see [8] for details.

Ideally, all colour-related definitions are specified in a scaleable way, so we can easily adjust a model, here the spatial resolution, by just changing a few constants. In the diffusion example, we have just one scaling constant, which



constants:
int $D = 10$; // for example
int $D1 = D$;
int $D2 = D$;
int $D3 = D$;
int MID=D/2;
colorsets:
$xdim = \{1..D1\}$;
$ydim = \{1..D2\}$;
$zdim = \{1..D3\}$;
$Grid3D = PROD(xdim \times ydim \times zdim)$;
variables:
x, a: $xdim$; y, b: $ydim$; z, c: $zdim$;

```
bool IsNeighbour(xdim x,ydim y,zdim z,xdim a,ydim b,zdim c) {
     (a=x+1 & b=y   & c=z ) | (a=x-1 & b=y   & c=z)
   | (a=x   & b=y+1 & c=z)  | (a=x   & b=y-1 & c=z)
   | (a=x   & b=y   & c=z+1)| (a=x   & b=y   & c=z-1) };
```

Fig. 1. Diffusion in 3D space as $\mathcal{CPN}^{\mathcal{C}}$. The transition guard (given in square brackets) determines by means of the colour function *IsNeighbour()*, if (a,b,c) is one of the six neighbours of (x,y,z).

is D, the size of the spatial cube. By unfolding we obtain a $\mathcal{CPN}$ made of D^3 places, $6 \cdot D^3 - 6 \cdot D^2$ transitions, and twice as many arcs.

Coloured Petri nets can be constructed from uncoloured Petri nets by folding, when the partitions of places and transitions are given. Then, these partitions define the colour sets of the coloured net. Vice versa, coloured Petri nets with finite colour sets can be automatically unfolded into uncoloured Petri nets, which then allows the application of all analysis and simulation techniques and related tools available for the corresponding unfolded Petri net class. See [26] for a formal definition of the unfolding problem. In the next section we show how unfolding can take advantage of symbolic data structures.

3 Interval Decision Diagrams

We first briefly recall in an informal way the core principles of Interval Decision Diagrams (IDD) as applied in this paper; see [31] for more details and formal definitions. Next, we illustrate the IDD-based unfolding by a simple example.

3.1 General Principles

IDDs have been first proposed in [21] and [33], probably independently. IDDs belong to the symbolic data structures and can be seen as a generalisation of the popular Binary decision diagrams (BDD). BDDs became a widely used data structure to encode Boolean functions, while IDDs encode interval logic functions, induced by expressions of the interval logic, originally defined in [21] to describe marking sets of P/T nets. Interval logic functions are, loosely speaking, Boolean expressions involving atomic predicates defining integer intervals, such as $x_1 \in [6, 8)$ or $x_2 > 0$.

Like BDDs, IDDs are Directed acyclic graphs (DAGs) with two types of nodes – terminal and non-terminal ones. There are two terminal nodes (typically represented as boxes), labelled with 0 and 1, and the non-terminal nodes (typically represented as circles or ellipses) are labelled with the variables occurring in the interval logic function to be encoded. These variables have to be totally ordered, i.e., they occur in the same order and at most once along each path from the root to one of the two terminal nodes.

In contrast to BDDs, non-terminal nodes in IDDs may have an arbitrary number of outgoing arcs labelled with intervals of natural numbers (including zero) partitioning the set of natural numbers. We consider intervals which have the form $[a, b)$; the lower bound a is included in the interval $[a, b)$, the upper bound b not. Note that intervals of the form $[a, \infty)$ are allowed as well, see Fig. 2 for two examples.

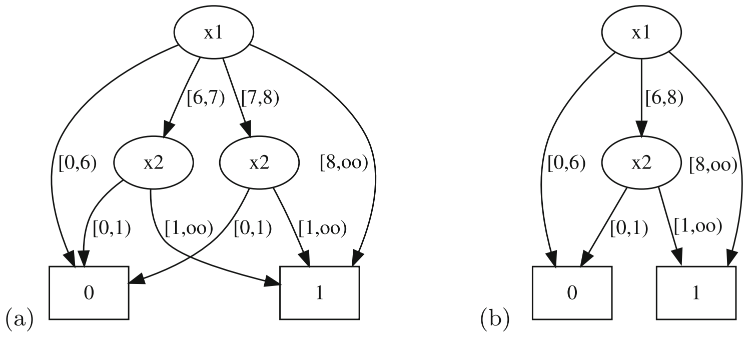


Fig. 2. Two IDDs representing $f = (x_1 \geq 8) \vee (x_1 \in [6, 8) \wedge x_2 > 0)$; **(a)** not reduced; **(b)** reduced.

Reduced ordered interval decision diagrams (ROIDD) are a canonical representation for interval logic functions and often provide a compact representation in many application areas. An IDD is called reduced if three conditions hold.

– The interval partitions labelling the outgoing arcs of each non-terminal node are reduced.
– Each non-terminal node has at least two different children.
– There exist no two nodes with isomorphic subgraphs.

The IDD in Fig. 2(a) is obviously not reduced. The third rule asks to merge the two nodes labelled with $x2$, which yields, having applied the first rule as well, the ROIDD in Fig. 2(b). These reduction rules are a straightforward generalisation of the rules for reduced ordered BDDs (ROBBD). As IDDs are a generalisation of BDDs, it does not come as a big surprise that the same issues hold with respect to the variable ordering.

- The variable ordering can have a great impact on the size of an ROIDD.
- In general, finding an optimal ordering is infeasible, even checking if a particular ordering is optimal is NP-complete.
- There exist interval logic functions that have ROIDD representations of exponential size for any variable ordering.
- Heuristics taking into consideration that variables which depend on each other should be close together in the ordering bring often good results.

Nevertheless, IDDs allow to define and implement efficient algorithms for the manipulation of interval logic functions [31,34]. All our algorithms use shared ROIDDs, an implementation principle to keep several ROIDDs within one data structure. Technically speaking, a shared ROIDD is a single multi-rooted DAG representing a collection of interval logic functions. All functions in the collection must be defined over the same set of variables, using the same variable ordering. Thanks to the canonicity of ROIDDs, two functions in the collection are identical if and only if the ROIDDs representing these functions have the same root in the shared ROIDD.

Our IDD implementations often perform fairly well, as we have seen for Marcie's model checkers for CTL [12,34] and CSL [11,30,32]. Thus, we will explore next how IDDs may possibly be of help for the unfolding problem.

3.2 IDD-Based Unfolding

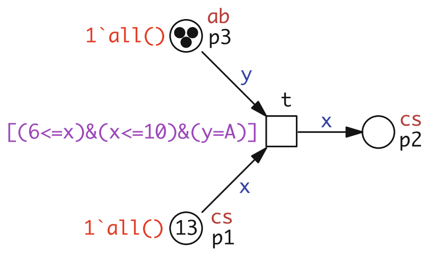
All our unfolding engines proceed basically in three steps.

1. *Unfolding of coloured places* – generates for each coloured place as many unfolded places as we have colours in the place's colour set, which is also reflected in the applied naming convention for the generated unfolded places. If the initial marking of a coloured place p contains n tokens of the colour c, then the unfolded place p_c has initially n (black) tokens.
2. *Unfolding of coloured transitions* – generates an unfolded transition (transition instance) for every variable binding and connects this unfolded transition with those unfolded places which correspond to the binding. The naming convention for the generated unfolded transition reflects the variable binding.
3. *Deleting any isolated unfolded places* – Colours which are never used yield isolated places, which will never influence the net behaviour, even if initially holding tokens; thus they can be safely removed.

The first and last step are relatively easy. The core problem of efficient unfolding is to determine the transition instances, i.e. all bindings of values to the variables involved, potentially enabling coloured transitions. Fortunately, we can consider each coloured transition t separately, and the problem can be formulated as a constraint satisfaction problem (CSP), defined by

- *the set of variables* – all variables occurring on any arc adjacent to t;
- *the domain of each variable* – given by its (finite, discrete) colour set;
- *the constraints* – any guards involved, which are all Boolean expressions.

To solve the CSP, we build step-wise bottom-up a corresponding IDD, the so-called *constraint IDD*. First, the domain of each individual variable is represented as IDD; the only colour set type causing here problems is *union*. Next, the constraint IDD is constructed using standard IDD algorithms. The set of all paths going from the root to the terminal node 1 describes all solutions of the given constraint problem; typically, one path encodes more than one solution. Thus, we can easily pick all CSP solutions from the constraint IDD.



colorsets:
cs = {1,8,3..6,10,9,11,20..23};
enum ab = {A,C,D};

variables:
cs : x;
ab : y;

Fig. 3. A simple coloured Petri nets.

For illustration let's consider the simple coloured Petri net given in Fig. 3. The stepwise computation of the constraint IDD to find all instances for the single transition t is documented in Fig. 4 and comprises the following steps.

(a) Encoding the entire colour set cs, specified in the coloured Petri net by the (deliberately) unordered list $\{1, 8, 3..6, 10, 9, 11, 20..23\}$. The IDD's basic principles yield automatically an ordered list; compare all paths going to the terminal node 1.

(b) Constraining the colour set cs given in (a) to $6 \leq x$ yields the subrange comprising $\{6, 8..11, 20..23\}$.

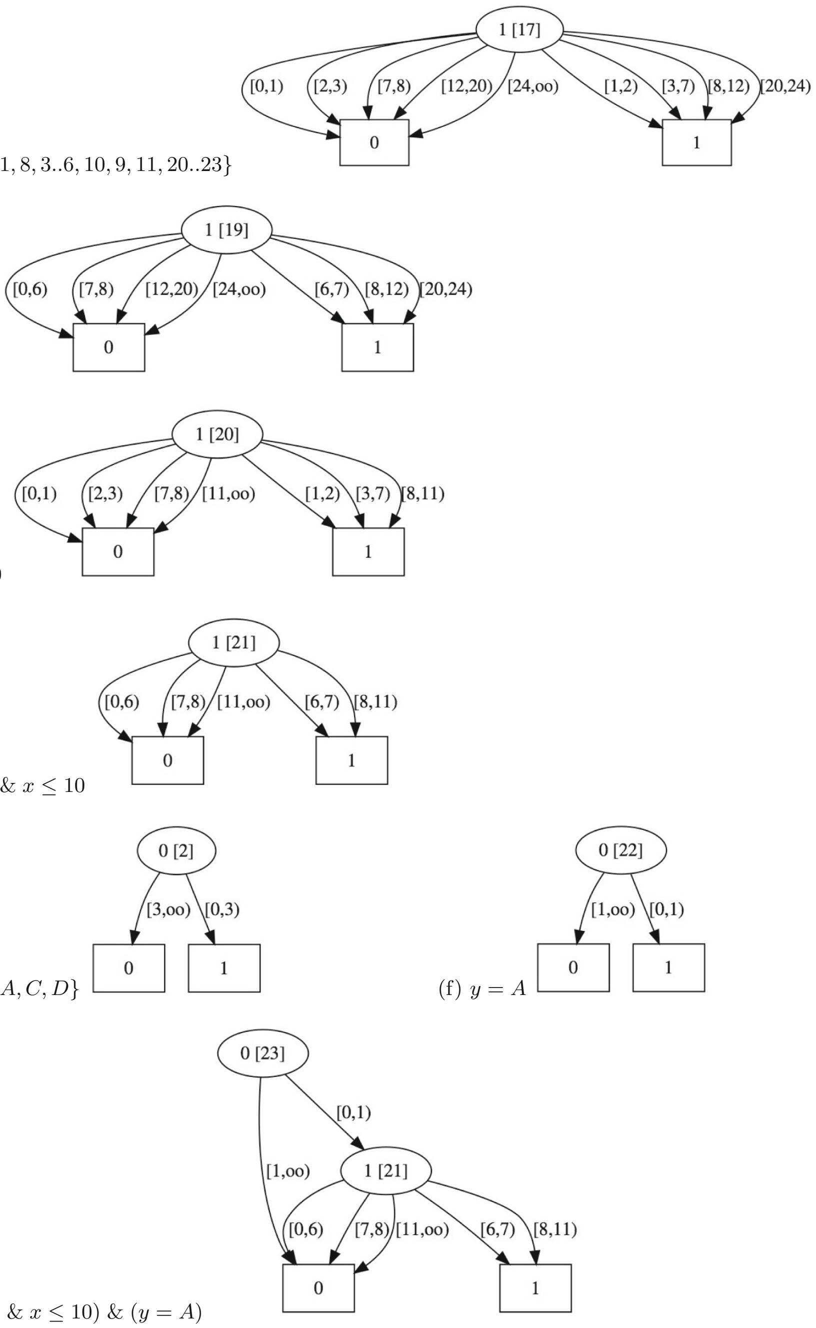
(c) Constraining the colour set cs given in (a) to $x \leq 10$ yields the subrange comprising $\{1, 3..6, 8..10\}$.

(d) Combining (b) and (c) by the logical operator & yields the IDD for the interval logical expression $6 \leq x \;\&\; x \leq 10$; the corresponding subrange now comprises $\{6, 8..10\}$.

(e) Encoding the entire enumeration colour set $ab = \{A, C, D\}$ automatically involves a mapping of the enumerated constants to integer identifiers; thus, the integer identifier 0 represents A, 1 stands for C, and 2 for D.

(f) Constraining the colour set ab given in (e) to $y = A$ yields the subrange comprising the single value A, represented by its identifier 0.

(g) The final result is computed by combining (d) and (f) by the logical operator & defining all possible bindings for the transition t according to its guard



(a) $cs = \{1, 8, 3..6, 10, 9, 11, 20..23\}$

(b) $6 \leq x$

(c) $x \leq 10$

(d) $6 \leq x \;\&\; x \leq 10$

(e) $ab = \{A, C, D\}$

(f) $y = A$

(g) $(6 \leq x \;\&\; x \leq 10) \;\&\; (y = A)$

Fig. 4. Stepwise IDD computation to find all instances (bindings) for the single transition t of the coloured Petri net given in Fig. 3.

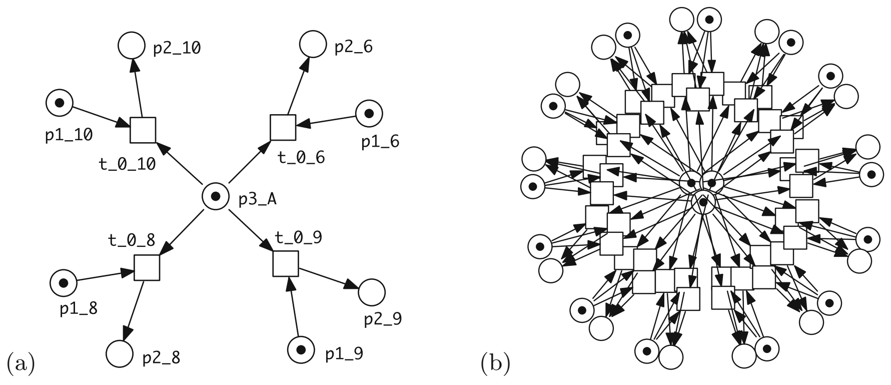


Fig. 5. **(a)** The unfolded Petri net corresponding to the coloured Petri net in Fig. 3. Transition bindings determined by the constraint IDD given in Fig. 4(g). **(b)** The unfolded Petri net for the coloured Petri net in Fig. 3 without transition guard. Layouts automatically generated by Snoopy's built-in layout library.

$(6 \leq x \ \& \ x \leq 10) \ \& \ (y = A)$. The (automatically) chosen variable order is $y \succ x$. There are two paths going to the terminal node 1. The first path encodes the value binding (A,6), and the second path encodes three value bindings: (A,8), (A,9), (A,10). Thus, in total we obtain four bindings, giving us four uncoloured transition instances for the coloured transition t.

All IDDs in Fig. 4 have been generated by a logging mechanism integrated in the unfolding engine for debugging purposes and visualised with Graphviz [4]. Non-terminal nodes are labelled with the variable index and the node number in the shared IDD (in square brackets).

Initial Marking. In Fig. 3, the two coloured places $p1$ and $p3$ are initialised with one token of each colour of the places' colour sets (by help of the pre-defined function all()). Thus, all places which we obtain by unfolding $p1$ and $p3$ are initialised with one token, and all unfolded places obtained by unfolding $p2$ remain empty. The finally generated unfolded Petri net is given in Fig. 5(a).

If the guard in the coloured Petri net in Fig. 3 were empty (meaning being simply *true*), then the cross product of the colour sets of x and y would be computed, and the unfolded Petri net would comprise $2 \cdot 13 + 3$ places and $13 \cdot 3$ transitions, each having three adjacent arcs; compare Fig. 5(b).

To deal with variables of union type, we need to consider – alternatively – all different data types subsumed by the union type, each yielding one constraint IDD. With other words: if we have two variables of a union type subsuming three types (colour sets), we obtain the solution by considering nine constraint IDDs. This might explain somehow why the union type is not really popular (anymore) in living programming languages.

As we have seen, guards define constraints. Guards, which may be arbitrarily complex, may not only serve as transition guards, as in the example just discussed, but also help to conveniently define colour sets as subsets of previously defined colour sets or to specify the initial marking in a concise and scalable way. Both needs to be considered when unfolding places. Likewise, guards also permit to specify colour-dependent transition rate functions or the arcs' conditional colour expressions. The algorithms given in the next section will take care of all of them.

4 Algorithms

This section sketches our implementation of the IDD unfolding engine by a pseudocode description; see Algorithms 1–4. The pseudocode is meant to be self-explanatory; we spend a few complementary remarks anyway.

Algorithm 1. The main procedure of the IDD unfolding engine follows the basic steps outlined previously in Sect. 3.2. But before unfolding the coloured places (line 17) and unfolding the coloured transitions (line 18), all colour-related net annotations have to be registered (lines 5–16). This comprises four categories of declarations: constants, colour sets, variables, and colour functions. Constants are crucial to design scalable and easily adjustable coloured Petri nets; thus they are often used in colour sets and colour functions.

Algorithm 1. Unfold CPN

```
1  Net unfoldedNet
2  placeRefTable $\subset$ String $\times$ Int $\times$ Int $= \emptyset$ // (name, tokens, number of references)
3  Environment env // some kind of registry
4  proc UNFOLDNET (CPN net)
5      forall $c \in$ net.constants do
6          env.registerConstant(c.name, c.expr)
7      od
8      forall $cs \in$ net.colorsets do
9          env.registerColorset(cs.name, cs.expr)
10     od
11     forall $v \in$ net.variables do
12         env.registerVariable(v.name, v.colorset)
13     od
14     forall $cf \in$ net.colorfunctions do
15         env.registerColorFunction(cf)
16     od
17     unfoldPlaces(net) // Algorithm 2
18     unfoldTransitions(net) // Algorithm 3
19     forall (place, tokens, ref) $\in$ placeRefTable do
20         if $ref > 0$ then unfoldedNet.addPlace(place,tokens) fi
21     od
22 end
```

The actual unfolding happens in Algorithms 2 and 3, which involves setting up and solving a CSP for every place and every transition, respectively. This is here done by help of IDDs, but could be equally achieved by any other appropriate data structure. Algorithm 2 creates unfolded places, but does not add them to the unfolded net. Algorithm 3 creates unfolded transitions and their unfolded adjacent arcs and does indeed add them to the unfolded net. Afterwards, all unfolded places, which are involved in the unfolding of transitions, are actually added to the unfolded net in the final step (lines 19–21), which implicitly prunes the unfolded net by ignoring isolated places.

Algorithm 2. Unfold Places

```
1  proc unfoldPlaces ($CPN\ net$)
2    forall $p \in net.places$ do
3      substituteColorFunctions(p.markingExpr, env) // replace fun call by its body
4      // guard used to describe subsets
5      $Guard\ g_{cs} = env$.implicitGuard(p.colorset)
6      $Set\ G_p = \{g_{cs}\}$
7      forall $expr \in p.markingExpr$ do// separated by '++'
8        $Set\ vars$ = collectVariables($markingExpr$, $env$)
9        $IDDSolutionSpaceRepr\ S(vars, expr.guard, env)$
10       createPlaces($p$, $S$, $expr.value$, $expr.color$)
11       $G_p = G_p \cup \{expr.guard \cap g_{cs}\}$
12     od
13     // remaining places are empty
14     $IDDSolutionSpaceRepr\ S(vars, \bigcap_{g \in G_p} \neg g, env)$
15     createPlaces(p, S, 0, expr.color)
16   od
17 end
18
19 proc createPlaces ($Place\ p$, $IDDSolutionSpaceRepr\ S$, $ColExpr\ value$, $ColExpr\ color$)
20   forall $sol \in S$ do
21     $place_s$ = createPlace($p$, $color$, $sol$, $env$)
22     $value_s$ = createValue($value$, $sol$, $env$)
23     $placeRefTable = placeRefTable \cup \{(place_s, value_s, 0)\}$
24   od
25 end
```

Algorithm 2. The unfolding of places can be done place by place and requires to determine all colours of a place's colour set. Thus, the computational load for this unfolding step depends on the kind of colour sets supported. Colour sets known by our tool family include the following.

– *Integer sets.* All colour sets are based on integer sets (to be precise: natural numbers). An integer colour set can be specified by a set of single elements or valid ranges, and may incorporate the usual set operations.
– *Enumeration types* are treated as integer sets, where all elements are given by constants.
– *Product sets.* Building on previously defined colour sets more complex, compound colour sets can be defined by means of the Cartesian product.
– *Subsets.* Given a previously defined colour set, it is possible to select specific elements characterised by a Boolean expression (*guard*). These guards are treated as implicit guards during the unfolding (line 5).

The computation of all colours for the colour set of a given place is achieved by constructing an IDD for the solution space (lines 9, 14). We obtain the solutions by following all path to the IDD's terminal node 1 (supported by a corresponding iterator concept); each solution generates an unfolded place (lines 20–24).

The creation of unfolded places includes the generation of their initial marking according to the given marking expression (lines 7–12). Places which remain empty are created afterwards (lines 14–15). Please note, places are created, but not added yet to the unfolded net.

Algorithm 3. The unfolding of transitions can be done transition by transition and requires to determine for every transition all variable bindings. To set up the corresponding CSP, the algorithm first iterates over all adjacent arcs (line 5), which are grouped into *conditions* (read arcs, inhibitory arcs, equal arcs, reset arcs) and *updates* (standard arcs connecting pre- and post-places). A transition guard may be additionally restricted by the implicit guards of any adjacent places with a subset colour set. Thus, those implicit guards have to be collected (lines 6–9). Next, all variables involved in any adjacent arc or transition guard

Algorithm 3. Unfold Transitions

```
1  proc UNFOLDTRANSITIONS ($CPN\ net$)
2    forall $t \in net.transitions$ do
3      $Guard\ G_a = \emptyset$
4      // preparation step;
5      forall $(p, arcType, arcExpr) \in t.conditions \cup t.updates$ do
6        // guard used to describe subsets
7        $Guard\ g_p = env$.implicitGuard($p.colorset$)
8        substituteColorFunctions($arcExpr, env$) // replace fun call by its body
9        $G_a = G_a \cup \{arcExpr.guards \cap g_p\}$
10     od
11     $Set\ vars$ = collectVariables($t.conditions \cup t.updates \cup t.guard, env$)
12     $IDDSolutionSpaceRepr\ S(vars, t.guard \cap (\bigcup_{g \in G_a} g), env)$
13     // creation step
14     forall $sol \in S$ do
15       $Set\ arcs$
16       forall $(p, arcType, arcExpr) \in t.conditions \cup t.updates$ do
17         $arc$ = createArc($p, arcType, sol, arcExpr.guard, arcExpr.value, arcExpr.color$)
18         if $arc \neq$ null then $arcs = arcs \cup \{arc\}$ fi
19       od
20       if $arcs \neq \emptyset$ then $unfoldedNet$.addTransition(createTransition($t, sol, env$),$arcs$) fi
21     od
22   od
23 end
24
25 func CREATEARC ($Place\ p, ArcType\ arcType, Solution\ sol, Guard\ guard, ColExpr\ value, ColExpr\ color$)
26   // guard used to describe subsets
27   $Guard\ g_p = env$.implicitGuard($p.colorset$)
28   if $sol \models guard \cap g_p$ then
29     $places_s$ = createPlace($p, color, sol, env$)
30     $values_s$ = createValue($value, sol, env$)
31     $placeRefTable = placeRefTable - \{(places_s, values_s, n)\} \cup \{(places_s, values_s, n+1)\}$
32     return $Arc(places_s, arcType, values_s)$
33   else
34     return null
35   fi
36 end
```

are collected (line 11), which then permits to create the IDD representation of the solution space of the given CSP (line 12).

Next, the CSP solutions are evaluated by iterating over the solution space, i.e., following all path to the IDD's terminal node 1 (lines 14–21). Every solution generally generates a set of arcs, whereby the unfolding of arcs always preserves the arc type; e.g., a coloured read arc will always be unfolded to read arcs. If there are no arcs for a given CSP solution, no unfolded transition is created (line 20).

Unfolded places will be ignored in Algorithm 1, if they are never connected to any transition. Thus, the entry in the *placeRefTable* is updated by removing the previous tuple and adding a new tuple with the number of references (i.e., usage of this place) increased by 1 (line 31).

Algorithm 4. This pseudocode is actually a data structure and provides the algorithm to construct an IDD representing the solution space for a given CSP,

Algorithm 4. IDD solution space representation

```
1  struct IDDSolutionSpaceRepr // Class
2    IDD solutions
3    proc constructor(Set vars, Guard guard, Environment env)
4      createVariableOrder(vars, guard)
5      solutions = 1 // the universe
6      // create the potential solution space
7      forall $v \in vars$ do
8        Intset is = env.getIntColorset(v)
9        solutions = solutions $\cap$ makeIDD(is,env)
10     od
11     // create the actual solution space
12     solutions = solutions $\cap$ makeIDD(guard,env)
13   end
14
15   func makeIDD(Guard g, Environment env)
16     if $g \equiv g_1 \wedge g_2$ then return makeIDD($g_1$,env) $\cap$ makeIDD($g_2$,env) fi
17     if $g \equiv g_1 \vee g_2$ then return makeIDD($g_1$,env) $\cup$ makeIDD($g_2$,env) fi
18     if $g \equiv \neg g_1$ then return 1− makeIDD($g_1$,env) fi
19     if $g \equiv f_1 \circ f_2 : \circ \in \{=, \neq, \leq <, >, \geq\}$ then
20       Set vars = collectVariables(f1) $\cup$ collectVariables(f2)
21       IDD S = 0 // empty set
22       forall $v \in vars$ do
23         Intset is = env.getIntColorset(v)
24         S = S $\cup$ makeIDD(is,env)
25       od
26       return ExtractAP(S, g)
27     fi
28   end
29 end
```

characterised by a set of variables and a guard in the context of the coloured net to be unfolded. The algorithm starts with choosing the variable order (line 4). A good variable order often depends on the specific guard involved; thus the guard occurs as parameter and is evaluated by the procedure *createVariableOrder* to determine which variables are close to each other. Next, the potential solution space is constructed by combining the colour sets of all variables involved (lines 7–10), which is afterwards restricted to the actual solution space by considering the guard (line 12). The actual IDD construction (lines 15–28) follows the standard IDD algorithms, see [31,32]. ExtractAP (Extract atomic proposition, line 26) extracts all states in S fulfilling the guard g, represented as IDD; for details see [32], Algorithm 6.

Our implementation is equipped with an iterator enabling the efficient iteration over all solutions, which is used in Algorithms 2 and 3. As a special feature, the iterator automatically updates the environment only with regard to changed variable values.

This algorithm is not IDD-specific. One could replace the type IDD just by some set type and the algorithm will work. Although IDDs often yield a very compact representation of sets and permit very efficient manipulation algorithms, it may be worth considering explicit or other symbolic data structures.

5 Experiments

The IDD-based unfolding is integrated in our `dssd_util` library used by Snoopy, Marcie and Spike. This section compares its runtime performance with our Gecode unfolding engine, i.e., with an unfolding engine employing the off-the-shelf library Gecode, which is a C++ toolkit for developing constraint-based systems [6]. Both unfolding engines are supported by our command-line tool Marcie. All experiments were done running scripts on a Linux machine in our computer lab (2.83 GHz, 32 GB RAM, running Ubuntu 18.04 LTS).

There is a growing library of (qualitative) benchmark models gathered over the years by the Petri net community for the yearly Model Checking Contest (MCC) [15]; among them are 22 scaleable coloured PNML models, for which different model versions are provided. We run our scripts over all of them. There is one model (BART) exceeding Marcie's unfolding skills; we did not get any results at all. All other models can be clearly classified into two categories.

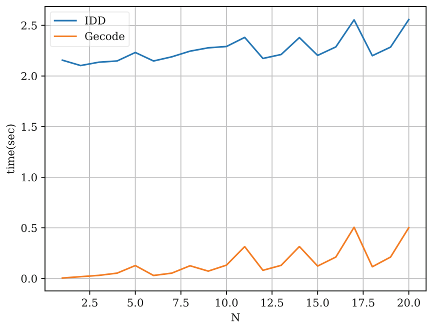
- *No substantial runtime.* There are 13 $\mathcal{PN}^{\mathcal{C}}$ with no substantial runtime for any model version; some even require a runtime below 1 sec, i.e., below a reliable measuring accuracy. A good example to illustrate this category is the *Bridges and vehicles* model.
- *Exponential runtime increase.* There are 8 $\mathcal{PN}^{\mathcal{C}}$ yielding all qualitatively equal results, which is best seen in the *Family reunion* model.

To cover different application scenarios we extend our benchmark suite by two examples from our own case study collection; both are quantitative $\mathcal{PN}^{\mathcal{C}}$ modelling biological processes and are scaleable with regard to some constant(s).

Thus, we consider here the following four models for our performance experiments.

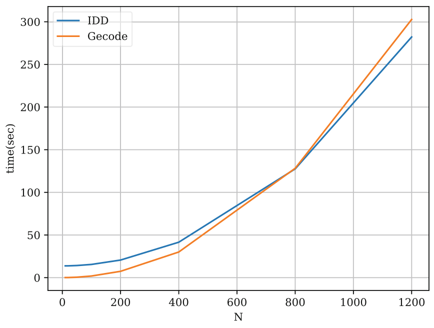
- **Bridges and Vehicles (MCC).** This $\mathcal{PN}^\mathcal{C}$ models an one-lane bridge of limited capacity, used by two types of vehicles. $\mathcal{PN}^\mathcal{C}$ size: 15 places, 11 transitions, and 57 arcs. There are three scaling parameters: the number of vehicles in each class, the bridge capacity, and the maximum number of vehicles of the same type being allowed to pass the bridge; see MCC website for details. There are 20 model versions, none requires substantial runtime, the entire experiment is completed in less than a minute; compare Fig. 6.
- **Family Reunion (MCC).** This $\mathcal{PN}^\mathcal{C}$ models the reunification process, which legal permanent residents have to follow to reunite with their families. $\mathcal{PN}^\mathcal{C}$ size: 104 places, 66 transitions, and 198 arcs. The model is parameterised by the number of legal residents, which in turn govern four further parameters; see MCC website for details. The unfolding time is short for smaller model versions, but quickly explodes for larger model versions; compare Fig. 7.
- **Diffusion in 3D.** Diffusion is a basic mechanism underlying many biological processes and is thus crucial for many case studies undertaken by help of our unifying framework, building on scaleable $\mathcal{SPN}^\mathcal{C}$, $\mathcal{CPN}^\mathcal{C}$ or $\mathcal{HPN}^\mathcal{C}$, as introduced in [8]. Here, we consider the $\mathcal{CPN}^\mathcal{C}$ given in Fig. 1, where each grid element in the 3D cube has six neighbours. The scaling factor is the grid size, i.e., the edge length of the 3D cube; compare Fig. 8.
- **Brusselator.** The Brusselator [29] is a popular reaction diffusion system, typically modelled as partial differential systems (PDE). Using this case study, we have shown in [23] how to systematically encode PDEs as $\mathcal{CPN}^\mathcal{C}$, which then can be equally read as $\mathcal{SPN}^\mathcal{C}$ or $\mathcal{HPN}^\mathcal{C}$. The coloured Petri net modelling the Brusselator involves diffusion in 2D with four neighbours and yields exactly the same results as obtained in [29] using PDEs. $\mathcal{CPN}^\mathcal{C}$ size: 2 places, 6 transitions, and 11 arcs. The nodes can either be read as discrete or continuous nodes. However, this does not have any effect on the unfolding efficiency. The scaling factor is the edge length of the 2D grid; compare Fig. 9.

Discussion. The IDD engine seems to come with some initial overhead; thus it is not advisable to use it for models, not requiring some substantial runtime (more than a few seconds), see Fig. 6. This also applies for the smaller versions of all models with an exponential increase of the runtime. But IDD unfolding outperforms Gecode unfolding for larger model versions, and we observe clear margins with increasing runtime, see Fig. 7. The margins in favour of the IDD unfolding turn out to be specifically dramatically for those case studies involving diffusion, see Figs. 8 and 9. These results equally apply to all our corresponding biological case studies, see, e.g., [5,13,17,24,28], to mention just a few. Scaling is crucial for case studies modelling biological processes evolving in time and space as it brings a higher resolution of the final results. In contrast, the scaling of all other examples shown here is more an academic exercise to challenge the unfolding engines. Based on these experiments, we recommend to use Gecode unfolding for such models, using only fairly simple and smallish colour sets and



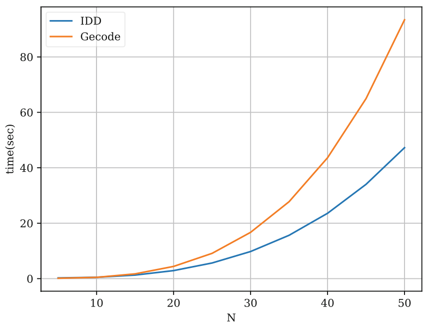
N	$\mid P \mid$	$\mid T \mid$	$\mid A \mid$
1	28	52	326
2	48	288	2090
4	78	968	7350
5	108	2228	17 190
9	128	1328	10 010
10	138	2348	18 090
11	168	5408	42 330
15	188	2108	15 950
16	198	3728	28 830
20	228	8588	67 470

Fig. 6. Bridges and vehicles (MCC); N – sequential model version number; Table shows selective results.



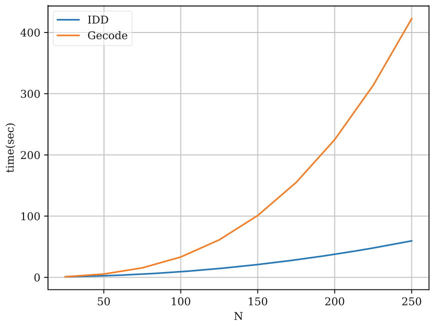
N	$\mid P \mid$	$\mid T \mid$	$\mid A \mid$
10	1475	1234	3799
20	3271	2753	8446
50	12 194	10 560	32 238
100	40 605	36 871	112 728
200	143 908	134 279	411 469
400	537 708	508 489	1 558 729
800	2 075 308	1 976 909	6 061 249
1200	4 612 908	4 405 329	13 507 769

Fig. 7. Family Reunion (MCC); N – legal residents.



N	$\mid P \mid$	$\mid T \mid$	$\mid A \mid$
5	125	600	1200
10	1000	5400	10 800
15	3375	18 900	37 800
20	8000	72 998	145 996
25	15 625	90 000	180 000
30	27 000	156 600	313 200
35	42 875	249 900	499 800
40	64 000	374 400	748 800
45	91 125	534 600	1 069 200
50	125 000	735 000	1 470 000

Fig. 8. 3D Diffusion with six neighbours; N – grid size of a 3D cube.



N	$\mid P \mid$	$\mid T \mid$	$\mid A \mid$
25	1250	11 908	23 191
50	5000	48 808	95 116
75	11 250	110 708	215 791
100	20 000	197 608	385 216
125	31 250	309 508	603 391
150	45 000	446 408	870 316
175	61 250	608 308	1 185 991
200	80 000	795 208	1 550 416
225	101 250	1 007 108	1 963 591
250	125 000	1 244 008	2 425 516

Fig. 9. Brusselator; N – grid size of a 2D square.

guards, and the IDD unfolding for models with more complex and larger colour sets and guards.

Reproducibility. All Petri nets used in our computational experiments are publicly available. The test cases taken from the MCC collection of coloured PNML files can be found at the MCC's website[1]. Our complementary set of test cases is provided as Snoopy and CANDL files[2]. In the same folder, one also finds a brief report documenting all computational experiments which we performed while writing this paper, and the Python scripts to repeat our computational experiments. All diagrams were generated with Python's Matplotlib [16].

6 Conclusions

In this paper we presented an approach building on symbolic data structures to efficiently unfold coloured Petri nets, qualitative and quantitative alike, into their plain, i.e. uncoloured counterparts. Our implementation builds on Interval Decision Diagrams and is available in the PetriNuts tool family for the modelling and simulation tool Snoopy, the analysis and simulation tool Marcie, and the simulation tool Spike, tailored to support reproducible simulation experiments.

We compared the runtime performance of the IDD unfolding engine with our Gecode unfolding engine; both are supported by Marcie. Our comparison incorporates all scalable coloured PNML models of the MCC benchmark suite. To the best of our knowledge, this has not been done before. We presented results of four case studies; two of the PNML benchmark suite and two of our own collection. There is no clear winner; which unfolding engine to use best seems to depend on the application scenario. The performance gain by IDD-based unfolding is specifically substantial for case studies involving non-trivial (implicit or explicit) guards. We appreciate that the exact runtime figures may

[1] https://mcc.lip6.fr/models.php.

[2] https://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Examples?dir=IddUnfolding.

depend on some implementation details, however we do generally not expect a dramatic change in the total order in which the two compared unfolding methods performed in our experiments. Due to the given page constraints, we had to confine ourselves to present four case studies only; results for more case studies are available as supplementary material on our website, including all scalable coloured PNML models of the MCC. Actually, Marcie reads PNML files; thus external users can test the IDD unfolding engine themselves without having to learn first Snoopy's coloured Petri nets or our exchange format CANDL.

Related work. Another idea to exploit symbolic data structures for the unfolding problem is to represent the unfolded net symbolically, as done in [19] using Data Decision Diagrams (DDD) [3]. First, the coloured net is unfolded in a straightforward manner and represented symbolically. Next, a couple of reductions of the unfolded net are performed on its symbolic representation. These reductions include the a posteriori removal of any false-guarded transitions, which should not have been generated in the first place. This requires, according to [19], some non-trivial DDD variable reordering and turned out to be less efficient than the traditional transition enabling test as done, e.g., by the tool Maria [27], which was one of the inspirational sources for our Generic unfolding engine. If required, the symbolic representation of the reduced unfolded net can be finally transformed into an explicit representation.

It would be interesting to explore whether our IDD-based unfolding strategy could be appropriately combined with a symbolic representation of the generated unfolded net. This would be specifically helpful if the analysis and simulation algorithms could directly deal with a symbolic net representation. Such a combination might help to efficiently deal with much larger unfolded nets as we are currently able to cope with.

Future Work. Useful extensions of the current functionality of coloured Petri nets in the PetriNuts tool family include place-dependent arc expressions (self-modifying $\mathcal{PN}^{\mathcal{C}}$), as exploited in [14] to capture cell growth and division in the eukaryotic cell cycle.

In our experiments, we compared the competing unfolding engines w.r.t. runtime, neither memory nor power consumption have been considered so far. This might not be appropriate for some specific use cases.

Envisaged performance-related improvements include:

– *Multi-threading exploiting state-of-the-art multi-core computers.* So far, the unfolding considers coloured places/transitions sequentially. one after the other. Obviously, this could be done in parallel, as each coloured place/transition defines its own CSP, which can be all solved independently. On top, the constraint IDD for a given CSP could be computed in a data-flow driven partial order.
– *Reuse of already computed solutions.* Often, variables, guards and colour sets are the same for several places and/or transitions. In this case, a recomputation of the actual solution space is a waste of time. Instead, the IDD unfolding should reuse previously computed IDD instances.

– *Choosing a variable order strategy.* It is well known that the efficiency of any algorithm relying on symbolic data structures generally depends on the chosen variable order; compare also Sect. 3.1. The current implementation deploys a simple dependency analysis of the guards. More sophisticated heuristics may be necessary for models with higher numbers of variables; we consider the option to explore some of the ordering strategies discussed in [1].

Acknowledgement. Fei Liu has been supported by National Natural Science Foundation of China (61873094), Science and Technology Program of Guangzhou, China (201804010246), and Natural Science Foundation of Guangdong Province of China (2018A030313338).

References

1. Amparore, E., Donatelli, S., Beccuti, M., Garbi, G., Miner, A.: Decision diagrams for Petri nets: which variable ordering? In: Proceedings of PNSE 2017. CEUR Workshop Proceedings, vol. 1846, pp. 31–50. CEUR-WS.org (2017)
2. Chodak, J., Heiner, M.: Spike – reproducible simulation experiments with configuration file branching. In: Bortolussi, L., Sanguinetti, G. (eds.) CMSB 2019. LNCS, vol. 11773, pp. 315–321. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31304-3_19
3. Couvreur, J.-M., Encrenaz, E., Paviot-Adet, E., Poitrenaud, D., Wacrenier, P.-A.: Data decision diagrams for Petri net analysis. In: Esparza, J., Lakos, C. (eds.) ICATPN 2002. LNCS, vol. 2360, pp. 101–120. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-48068-4_8
4. Gansner, E., North, S.: An open graph visualization system and its applications to software engineering. Softw.: Practice Experience **30**(11), 1203–1233 (2000)
5. Gao, Q., Gilbert, D., Heiner, M., Liu, F., Maccagnola, D., Tree, D.: Multiscale modelling and analysis of planar cell polarity in the drosophila wing. IEEE/ACM Trans. Comput. Biol. Bioinform. **10**(2), 337–351 (2013)
6. Gecode Team: Gecode: Generic constraint development environment. http://www.gecode.org
7. Genrich, H.J., Lautenbach, K.: The analysis of distributed systems by means of predicate/transition-nets. In: Kahn, G. (ed.) Semantics of Concurrent Computation. LNCS, vol. 70, pp. 123–146. Springer, Heidelberg (1979). https://doi.org/10.1007/BFb0022467
8. Gilbert, D., Heiner, M., Liu, F., Saunders, N.: Colouring space - a coloured framework for spatial modelling in systems biology. In: Colom, J.-M., Desel, J. (eds.) PETRI NETS 2013. LNCS, vol. 7927, pp. 230–249. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38697-8_13
9. Heiner, M., Herajy, M., Liu, F., Rohr, C., Schwarick, M.: Snoopy – a unifying Petri net tool. In: Haddad, S., Pomello, L. (eds.) PETRI NETS 2012. LNCS, vol. 7347, pp. 398–407. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31131-4_22
10. Heiner, M., Rohr, C., Schwarick, M.: MARCIE – model checking and reachability analysis done efficiently. In: Colom, J.-M., Desel, J. (eds.) PETRI NETS 2013. LNCS, vol. 7927, pp. 389–399. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38697-8_21

11. Heiner, M., Rohr, C., Schwarick, M., Streif, S.: A comparative study of stochastic analysis techniques. In: Proceedings of CMSB 2010, pp. 96–106. ACM Digital Library (2010)
12. Heiner, M., Rohr, C., Schwarick, M., Tovchigrechko, A.A.: MARCIE's secrets of efficient model checking. In: Koutny, M., Desel, J., Kleijn, J. (eds.) Transactions on Petri Nets and Other Models of Concurrency XI. LNCS, vol. 9930, pp. 286–296. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53401-4_14
13. Herajy, M., Liu, F., Rohr, C., Heiner, M.: Coloured hybrid Petri nets: an adaptable modelling approach for multi-scale biological networks. Comput. Biol. Chem. **76**, 87–100 (2018)
14. Herajy, M., Schwarick, M., Heiner, M.: Hybrid Petri nets for modelling the eukaryotic cell cycle. In: Koutny, M., van der Aalst, W.M.P., Yakovlev, A. (eds.) Transactions on Petri Nets and Other Models of Concurrency VIII. LNCS, vol. 8100, pp. 123–141. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40465-8_7
15. Hillah, L.M., Kordon, F.: Petri nets repository: a tool to benchmark and debug Petri net tools. In: van der Aalst, W., Best, E. (eds.) PETRI NETS 2017. LNCS, vol. 10258, pp. 125–135. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57861-3_9
16. Hunter, J.: Matplotlib: a 2D graphics environment. Comput. Sci. Eng. **9**(3), 90–95 (2007)
17. Ismail, A., Herajy, M., Heiner, M.: A graphical approach for hybrid modelling of intracellular calcium dynamics based on coloured hybrid Petri nets. In: Liò, P., Zuliani, P. (eds.) Automated Reasoning for Systems Biology and Medicine. CB, vol. 30, pp. 349–367. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17297-8_13
18. Jensen, K.: Coloured Petri nets and the invariant-method. Theoret. Comput. Sci. **14**(3), 317–336 (1981)
19. Kordon, F., Linard, A., Paviot-Adet, E.: Optimized colored nets unfolding. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 339–355. Springer, Heidelberg (2006). https://doi.org/10.1007/11888116_25
20. Kristensen, L., Christensen, S.: Implementing coloured Petri nets using a functional programming language. High.-Order Symb. Comput. **17**(3), 207–243 (2004)
21. Lautenbach, K., Ridder, H.: A completion of the S-invariance technique by means of fixed point algorithms. Technical report 10–95, Universität Koblenz-Landau (1995)
22. Liu, F.: Colored Petri nets for systems biology. Ph.D. thesis, BTU Cottbus, Department of CS (2012)
23. Liu, F., Blätke, M., Heiner, M., Yang, M.: Modelling and simulating reaction-diffusion systems using coloured Petri nets. Comput. Biol. Med. **53**, 297–308 (2014)
24. Liu, F., Heiner, M.: Multiscale modelling of coupled Ca2+ channels using coloured stochastic Petri nets. IET Syst. Biol. **7**(4), 106–113 (2013)
25. Liu, F., Heiner, M., Rohr, C.: Manual for colored Petri nets in Snoopy. Technical report 02–12, BTU Cottbus, Department of Computer Science (2012)
26. Liu, F., Heiner, M., Yang, M.: An efficient method for unfolding colored Petri nets. In: Proceedings of WSC 2012. 978-1-4673-4781-5/12. IEEE (2012). http://informs-sim.org
27. Mäkelä, M.: Optimising enabling tests and unfoldings of algebraic system nets. In: Colom, J.-M., Koutny, M. (eds.) ICATPN 2001. LNCS, vol. 2075, pp. 283–302. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45740-2_17

28. Pârvu, O., Gilbert, D., Heiner, M., Liu, F., Saunders, N., Shaw, S.: Spatial-temporal modelling and analysis of bacterial colonies with phase variable genes. ACM TOMACS **25**(2), 25p (2015)
29. Peña, B., Pérez-García, C.: Stability of Turing patterns in the Brusselator model. Phys. Rev. E **64**, 056213 (2001)
30. Schwarick, M., Heiner, M.: CSL model checking of biochemical networks with interval decision diagrams. In: Degano, P., Gorrieri, R. (eds.) CMSB 2009. LNCS, vol. 5688, pp. 296–312. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03845-7_20
31. Schwarick, M., Tovchigrechko, A.: IDD-based model validation of biochemical networks. Theoret. Comput. Sci. **412**(26), 2884–2908 (2011)
32. Schwarick, M.: Symbolic on-the-fly analysis of stochastic Petri nets. Ph.D. thesis, BTU Cottbus, Department of CS (2014)
33. Strehl, K., Thiele, L.: Symbolic model checking using interval diagram techniques. Technical report, Computer Engineering and Networks Lab (TIK), ETH Zurich (1998)
34. Tovchigrechko, A.: Efficient symbolic analysis of bounded Petri nets using Interval Decision Diagrams. Ph.D. thesis, BTU Cottbus, Department of CS (2008)