SYNOPS - Generation of Partial Languages and Synthesis of Petri Nets *

Robert Lorenz, Markus Huber, Christoph Etzel, and Dan Zecha

Department of Computer Science
University of Augsburg, Germany
robert.lorenz@informatik.uni-augsburg.de

Abstract. We present the command line tool SYNOPS. It allows the term-based construction of partial languages consisting of different kinds of causal structures representing runs of a concurrent system: labeled directed acyclic graphs (LDAGs), labeled partial orders (LPOs), labeled stratified directed acyclic graphs (LSDAGs) and labeled stratified order structures (LSOs). It implements region based algorithms for the synthesis of place/transition nets and general inhibitor nets from behavioural specifications given by such partial languages.

1 Introduction

Synthesis of Petri nets from behavioral descriptions has been a successful line of research since the 1990s. There is a rich body of nontrivial theoretical results and there are important applications in industry, in particular in hardware design [9,12], in control of manufacturing systems [25] and recently also in workflow design [23,22,1,10,4].

The synthesis problem is the problem to construct, for a given behavioral specification, a Petri net such that the behavior of this net coincides with the specified behavior (if such a net exists). There are many different methods which are presented in literature to solve this problem for different classes of Petri nets. They differ mainly in the Petri net class and the model for the behavioral specification considered. On the other hand, all these methods are based on one common theoretical concept, the notion of a *region* of the given behavioral specification.

In this paper, we present a new tool for the region based synthesis of Petri nets from behavioral specifications given by so called partial languages. A partial language is a set of finite causal structures, where a causal structure represents causal relationships between events of a finite run of a concurrent system. If the concurrent system is given by a Petri net, events represent transition occurrences. Expressible causal relationships are for example direct and indirect causal dependency, concurrency and synchronicity of events. The tool supports different kinds of causal structures, describing different semantics of different Petri net classes and having different expressiveness and interpretation:

* Supported by the German Research Council, Project SYNOPS 2008 - 2012 [13]

- Labelled acyclic graphs (LDAG): LDAGS represent runs underlying process nets of place/transition-nets. They are used to specify all direct causal dependencies caused by token flow between transitions occurrences.
- Labelled partial orders (LPO): LPOs represent non-sequential runs of place/transition-nets. They are used to specify all "earlier than"-relations (which we call indirect causal dependencies) between transitions occurrences. Unrelated events are called concurrent. LPOs are transtively closed LDAGs.
- LDAGs extended by synchronicity (LSDAG): LSDAGs represent runs underlying process nets of general inhibitor nets according to the a-priori-semantics. They are DAGs extended by "not later than"-relations between events. A cycle of "not later than"-relations between events represents a synchronous step of events, i.e. it is possible to distinguish between concurrency and synchronicity.
- Labelled stratified order structures (LSO): LSOs represent non-sequential runs of general inhibitor nets according to the a-priori-semantics. LSOs are transitively closed LSDAGs.

This means, by a partial language the set of runs of a Petri net for different Petri net classes and different net semantics can be specified. It depends on the application area, which Petri net class and which kind of causal structures are appropriate or available for solving a concrete synthesis problem. In [10,4] case studies are presented illustrating the applicability and usefulness of synthesis from partial languages in practise.

Infinite behaviour can be represented by an infinite set of finite runs, i.e. an infinite partial language (where one finite run can be the prefix of another finite run).

The tool allows to construct finite partial languages (allowing to specify finite behaviour) of the mentioned types via command line using a term-based notation. This term based notation allows to compose runs from a set of basic runs by several composition operators (sequential and parallel composition and iteration). For the synthesis of nets the tool implements algorithms based on a technique using so called token flows developed in the project SYNOPS [13]. Up to now, only an algorithm for the synthesis of place/transiton nets from finite sets of LPOs is supported.

The paper is organized as follows. In section 2 we briefly recall some basic mechanism of region-based synthesis. In section 3 we present some new technical developments for the synthesis of place/transition nets from finite sets of LPOs. In section 4 we describe the architcture and the components of the tool. In particular we describe how to specify finite sets of LPOs, LDAGs, LSDAGs and LSOs via command line. In section 5 we present some case studies involving the implemented algorithm for the synthesis of place/transition nets from finite sets of LPOs. In section 6 we briefly compare the tool to other synthesis tools. In section 7 we give a brief outlook onto current and further developments and in section 8 we give some hints for downloading and testing the tool.

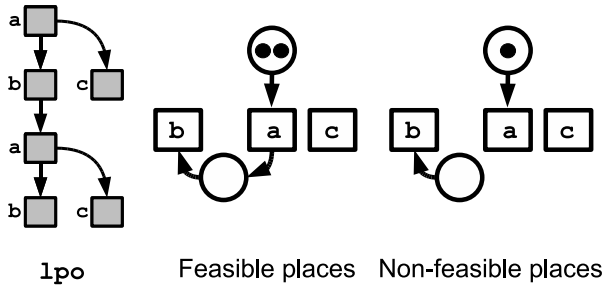

Fig. 1. An LPO (left part) and two feasible and two non-feasible places w.r.t this LPO.

2 Region based Synthesis

In this section, we denote the set of runs of a net N by $L(N)$. $L(N)$ is called the language generated by N. We formally consider the following synthesis problem w.r.t. different Petri net classes and different types of partial languages:

Given: A prefix-closed partial language L over a finite alphabet of transition names T.
Searched: A Petri net N with set of transitions T and $L(N) = L$.

That means, we search for an exact solution of the problem. Such an exact solution may not exist, i.e. not each language L is a *net language*.

The classical idea of region-based synthesis is as follows: First consider the net N having an empty set of places and set of transitions T. This net generates each run in L (i.e. $L \subseteq L(N)$), because there are no places restricting transition occurrences. But it generates much more runs. Since we are interested in an exact solution, we restrict $L(N)$ by adding places.

There are places p, which restrict the set of runs too much in the sense that $L \setminus L(N) \neq \emptyset$, if p is added to N. Such places are called *non-feasible (w.r.t. L)*. We only add so called *feasible* places p satisfying $L \subseteq L(N)$, if p is added to N (Figure 1). The idea of region-based synthesis is to add *all* feasible places to N. The resulting net N_{sat} is called the *saturated feasible net*. On the one hand, N_{sat} has by construction the following very nice property: $L(N_{sat})$ is the smallest net language satisfying $L \subseteq L(N_{sat})$. This is clear, since $L(N_{sat})$ could only be further restricted by adding non-feasible places. This property directly implies that there is an exact solution of the synthesis problem if and only if N_{sat} is such an exact solution. Moreover, if there is no exact solution, N_{sat} is the best approximation to such a solution "from above".

On the other hand, this result is only of theoretical value, since the set of feasible places is in general *infinite* (Figure 2). Therefore, for a practical solution, a finite subset of the set of all feasible places is defined, such that the net N_{fin} defined by this finite subset fulfills $L(N_{fin}) = L(N_{sat})$. Such a net N_{fin} is called

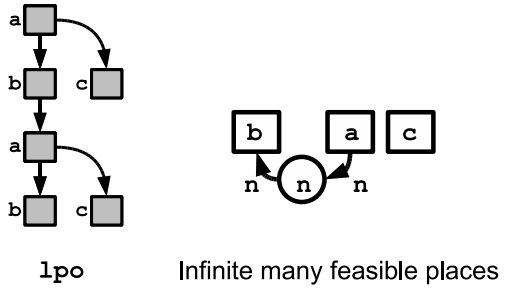

Fig. 2. The shown place is feasible w.r.t. the left LPO for each integer $n \in \mathbb{N}$.

finite representation of N_{sat}. In order to construct such a finite representation, in an intermediate step a feasible place is defined through a so called *region* of the given language L.

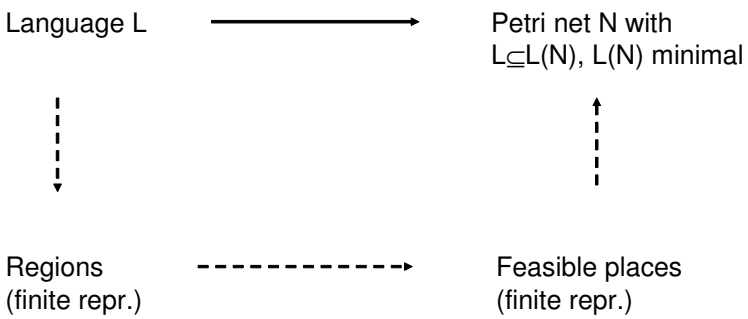

Fig. 3. The approach of region-based synthesis.

The described approach is common to all known region-based synthesis methods (see Figure 3) and can be applied to all kinds of partial languages. In particular, this approach can be applied to different notions of regions (of a partial language) and of finite representations N_{fin}. There are two types of definitions of regions and two types of definitions of finite representations, covering all known region-based synthesis methods [16].

Experiments in the first phase of the project SYNOPS showed that the so called separation representation produces Petri nets which are simpler and more compact, especially having less places [2]. Moreover, it turned out that so called token flow regions can be computed more efficiently in the presence of much concurrency. Therefore, the first synthesis algorithm implemented in SYNOPS computes a place/transition net from a finite set of LPOs using the separation representation of the set of all token flow regions. Note that this variant is not yet implemented in other tools.

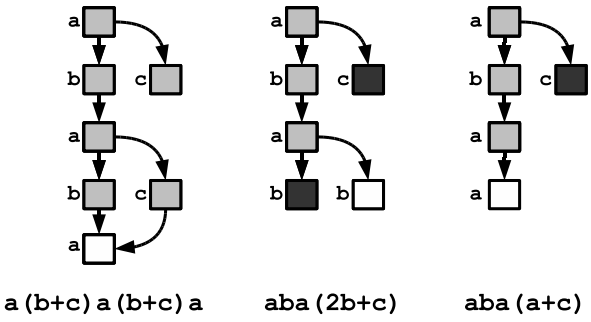

Fig. 4. Several wrong continuations of the LPO shown in the previous figures. A wrong continuation consists of a prefix (grey color) and a follower step (black) including an additional event (white) and represents one or more step sequences.

For computing the separation representation, first all so called wrong continuations of L are constructed. The set of wrong continuations represents the behaviour which is not specified. Briefly, a wrong continuation consists of a prefix of some specified run together with a follower step of transition occurrences extending a specified run by one additional event. Figure 4 shows examples of wrong continuations. For every wrong continuation, the synthesis algorithm tries to compute a place prohibiting the wrong continuation (for details on how to compute such a place we refer to [16]. The synthesized Petri net is an exact solution (does not have runs which are not specified) if and only if each wrong continuation can be prohibited by some place. Figure 5 shows the result of the synthesis algorithm. The wrong continuations shown in Figure 4 are forbidden by the places p_3, p_2 and p_1 (from left to right).

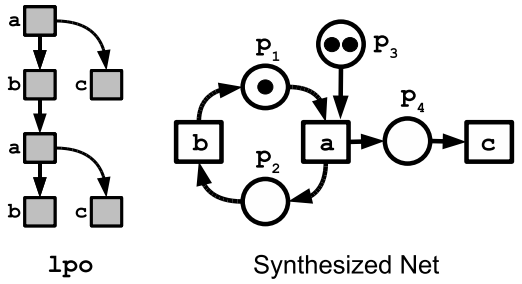

Fig. 5. Synthesized net (right part) for the partial language only containing the LPO shown in the left part.

The synthesized Petri net depends on the considered order of wrong continuations, since places often prohibit more than one wrong continuation. It is advantageous to compute such places first, which prohibit much wrong contin-

uations. Therefore several new methods were implemented for constructing an appropriate order of wrong continuations. In the next Section 3 some technical details of the implemented synthesis algorithm are described.

3 Newly developed Techniques

In this section we briefly introduce wrong continuations formally and describe some newly developed ideas optimizing the synthesis procedure.

A *multiset* over a set T is a function $m : T \to \mathbb{N}$. A *step* T is a multiset over T. Addition $+$ on multisets is defined by $(m + m')(a) = m(a) + m'(a)$. We write $\sum_{a \in T} m(a)a$ to denote a multi-set m. The relation $\leq$ between multiset is defined through $m \leq m' \iff \forall a \in T(m(a) \leq m'(a))$. An *LPO* over a set T is a tuple $(V, <, l)$ where V is the finite set of events, $<\subseteq V \times V$ is a partial order, and $l : V \to T$ is a labelling function. For $W \subseteq V$ we define the multiset $l(W)(a) = |\{v \in W \mid l(v) = a\}|$. An LPO $(W, <, l)$ is a *prefix* of an LPO $(V, <, l)$ if $W \subseteq V$ and $(w \in W) \wedge (v < w) \Rightarrow (v \in W)$. A step sequence $w = \alpha_1 \dots \alpha_n$ can be represented by an LPO, where each step α_i corresponds to a set of pairwise unordered events and events from different steps are ordered according to the step sequence. A step sequence σ is a *step linearization* of an LPO $(V, <, l)$, if the partial order representing σ contains $<$. For example, the step sequences $a(b + c)a(b + c)$, $ab(a + c)(b + c)$ and $aba(b + 2c)$ are step linearizations of the LPO shown in Figure 5.

Throughout this section, let L be a prefix closed partial language of LPOs. We denote L^{step} the set of all *step-linearizations* of LPOs in L. Since $lpo \in L$ is a run of a net N if and only if each step linearization of lpo is a step execution of N, wrong continuations are defined formally as step sequences which extend elements from L^{step} by one event as follows:

Definition 1 (Wrong Continuation). *Let $\sigma = \alpha_1 \dots \alpha_{n-1}\alpha_n \in L^{step}$ and $t \in T$ such that $w_{\sigma,t} = \alpha_1 \dots \alpha_{n-1}(\alpha_n + t) \notin L^{step}$, where α_n is allowed to be the empty step. Then $w_{\sigma,t}$ is called* wrong continuation of L.

We call $\alpha_1 \dots \alpha_{n-1}$ the prefix *and $\alpha_n + t$ the* follower step of the wrong continuation.

To prohibit a wrong continuation, one needs to find a feasible place p such that after occurrence of its prefix there are not enough tokens to fire its follower step. A prefix $\alpha_1 \dots \alpha_{n-1}$ of a wrong continuation stepwise linearizes a prefix of an LPO in L. A follower step of such a LPO-prefix can constructed by taking a subset of its direct successor in the LPO and add an event with a new label. This means, a wrong continuation can be represented on the level of LPOs, where wrong continuations having the same follower step and whose prefixes stepwise linearize the same LPO-prefix need not be distinguished. For example $a(b+c)a(b+c)a$, $aba(2b+c)$ and $aba(a+c)$ are wrong continuations of the LPO shown in Figure 5. Their representations on the level ofs LPOs are shown in Figure 4 (from left to right).

Since the follower marking after the occurrence of a prefix of a wrong continuation only depends on the number of occurrences of each transition (but not on their ordering), the following statement holds:

Proposition 1. *Let* $w_{\sigma,t} = \alpha_1 \dots \alpha_{n-1}(\alpha_n + t)$ *be a wrong continuation and* $\sigma' = \alpha'_1 \dots \alpha'_{m-1}\alpha_n \in L^{step}$ *satisfying* $\alpha_1 + \ldots + \alpha_{n-1} = \alpha'_1 + \ldots + \alpha'_{m-1}$. *Then* $w_{\sigma,t}$ *can be prohibited it and only if* $w_{\sigma',t}$ *can be prohibited.*

That means in particular, for storing the set of all wrong continuations it is enough to construct all pairs $(l(W), l(S))$, where $(W, <, l)$ is a prefix of some LPO in L and S is a subset of direct successors of $(W, <, l)$ extended by an additional event. For example, the wrong continuation $a(b+c)a(b+c)a$ is stored in the form $(2a+2b+2c, a)$. Moreover, the follower steps of wrong continuations with equivalent prefixes need to be merged.

We now define an order on the set of wrong continuations.

Definition 2 (More restrictive wrong Continuation). *A wrong continuation* $w_{\sigma,t}$ *is* more restrictive *than a wrong continuation* $w_{\sigma',t'}$, *if the following holds: If* $w_{\sigma,t}$ *is not a step execution of a place/transition net* N, *then* $w_{\sigma',t'}$ *is not a step execution of* N.

If it is possible to forbid a wrong continuation, then automatically all less restrictive wrong continuations are forbidden, too. This means, if one considers more restrictive wrong continuations first, then less places are computed and runtime is faster.

If two wrong continuations have equivalent prefixes and the follower step of the first is included in the follower step of the second, then the first wrong continuation is more restrictive than the second one, since its follower step needs less tokens. For example $a(b + c)a(2b)$ is more restrictive than $a(b + c)a(2b + c)$ in this sense.

Proposition 2. *Let* $w_{\sigma,t} = \alpha_1 \dots \alpha_{n-1}(\alpha_n + t)$ *and* $w'_{\sigma',t'} = \alpha'_1 \dots \alpha'_{m-1}(\alpha'_m + t')$ *be wrong continuations of* L *satisfying* $\alpha_1 + \ldots + \alpha_{n-1} = \alpha'_1 + \ldots + \alpha'_{m-1}$ *and* $(\alpha_n + t) \leq (\alpha'_m + t')$. *Then* $w_{\sigma,t}$ *is more restrictive than* $w_{\sigma',t'}$.

If the last step of a wrong continuation is sequenzialized by several terminal steps of a second wrong continuation, then the second wrong continuation is more restrictive than the first one, since a step is not enabled, if a sequentialization of the step is not enabled in a marking. For example $a(b + c)aa$ is more restrictive than $a(b + c)(2a)$ in this sense.

Proposition 3. *Let* $w_{\sigma,t} = \alpha_1 \dots \alpha_{n-1}(\alpha_n + t)$ *and* $w_{\sigma',t'} = \alpha'_1 \dots \alpha'_{m-1}(\alpha'_m + t')$ *be wrong continuations of* L *satisfying* $\alpha_1 + \ldots + \alpha_{n-1} + (\alpha_n + t) = \alpha'_1 + \ldots + \alpha'_{m-1} + (\alpha'_m + t')$ *and* $\alpha_1 + \ldots + \alpha_{n-1} \geq \alpha'_1 + \ldots + \alpha'_{m-1}$. *Then* $w_{\sigma,t}$ *is more restrictive than* $w_{\sigma',t'}$.

According to these oberservations, wrong continuations are ordered in the following way: Wrong continuations with longer prefixes are considered first and if two wrong continuations have equal prefix, then the wrong continuation with the shorter follower step is considered first.

4 Architecture and Functionality

4.1 Overview

The SYNOPS tool is implemented strictly following advanced object oriented paradigms using a classical 3-tier-architecture:

- The client tier is realized as a command line interface (`CLI`). In the meanwhile we also provide a graphical user interface (`GUI`) which additionally visualizes Petri net synthesis results. The `CLI` (resp. `GUI`) and the middle tier are loosly coupled, such that an easy and fast change is possible.
- The middle tier (`SynCore`) encapsulates data types for the supported kinds of runs, sets of such runs and Petri nets and basic operations for creating, manipulating and destroying such objects. It can only be accessed via a facade (`SynShell`).
- Sets of runs and synthesized Petri nets are stored in text files. For Petri nets the PNML-standard is used, such that synthesis results can be visualized by many Petri net editors. For storing sets of runs we use a simple self-created text format which lists runs, events and edges.

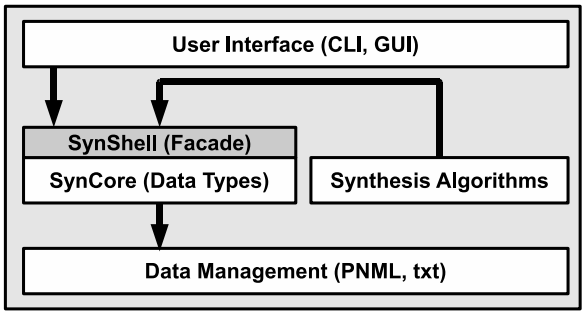

Fig. 6. Architecture of the system.

Figure 6 shows the architcture. Synthesis algorithms are connected with `SynCore` through a plug-in system, where each plug-in communicates with `SynCore` via the facade `SynShell`. `SynShell` implements interfaces supporting such a plug-in system.

4.2 The middle tier `SynCore`

A run consists of a finite set of events labelled by action names and a finite set of directed edges between events. A run can be represented through the four different causal structures previously mentioned.

The object of interest are sets of runs, since synthesis algorithms are operating on such sets. Every event has an ID which is unique within a run. Every run has

an ID which is unique within a set of runs. Sets have global unique IDs. This way, each object can be identified by a combination of IDs in the usual way. For example, the identifier `set1.lpo5.event3` represents the event with ID `event3` in the run with ID `lpo5` belonging to the set with ID `set1`.

There are several useful operations for manipulation of these data structures, for example operations testing consistency properties of runs specified by the user (such as cycle-freeness), operations computing the transitive closure of runs specified by the user, operations computing all prefixes of a run (based on a modified version of the algorithm of Warshall [24]) and operations computing the direct successors of a prefix of a run.

A Petri net consists of places, transitions and two kinds of edges between places and transitions (flow edges and inhibitor edges). Places have a unique ID, a name, a number of tokens and a maximum capacity of tokens (which can be infinity). Transitions have a unique ID and a name. Edges have a weight. This way several low level Petri net classes can be represented such as place/transition nets and inhibitor nets. Moreover, there are several restrictions available such as a bound of 1 for arc weights in order to represent elementary Petri nets. Such restrictions are realized by overwriting methods in specialized classes. This modular construction makes it easy to extend the framework by other net classes in future.

4.3 Synthesis algorithms

So far, there is only one synthesis algorithm implemented in the download version of the tool: The algorithm `syn-tf-sep` computes place/transition nets from finite sets of LPOs using the separation representation of the set of token flow regions.

4.4 Command line interface `CLI`

The `CLI` allows easy construction of long runs and sets of runs using a term-based notation. Currently, each command may only contain one operation. Complicated terms are constructed stepwise command by command.

A set (of runs) is opened by `'set ID'`. After opening a set, runs of the set can be specified. Runs can only be specified within a set. Finally, a set is closed by `'tes'`.

A run is opened by `'dag ID'` (for LDAGs), `'lpo ID'` (for LPOs), `'sdag ID'` (for LSDAGs) or `'lso ID'` (for LSOs). After opening a run, events and edges of the run can be specified. Events and edges can only be specified within a run. A run is closed by `'gad'`, `'opl'`, `'gads'` or `'osl'`. After closing a run, consistency of the user input is checked. Moreover, in case of LPOs and LSOs, the transitive closure is constructed (that means, it is not necessary to specify all transitive edges). Finally, all prefixes of the run are computed, preparing the synthesis computation.

An event is specified by `'event ID LABEL'`. An edge in a LDAG or an LPO between two events with IDs `e1` and `e2` is specified by `'et e1 e2'`. A "not later

than" edge is specified by `'nlt e1 e2'`. Using these operations, simple runs can be constructed such as LPO `lpo1` shown in Figure 7. Figure 8 shows the syntax for specifying `lpo1`.

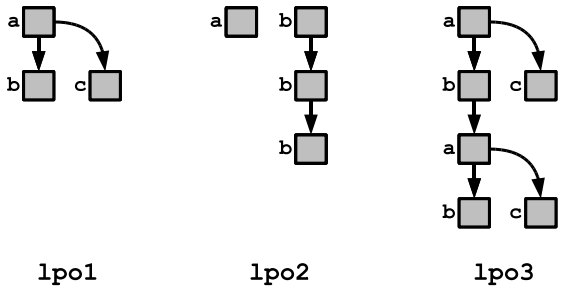

Fig. 7. Examples of LPOs.

```
1 set set1
2 lpo lpo1
3 event a a
4 event b b
5 event c c
6 et a b
7 et a c
8 opl
9 tes
```

Fig. 8. Syntax for specifying LPO `lpo1`.

There are several operations for combining existing runs:

- If `run1` and `run2` are runs, then by `'append ID run1 run2'` the sequential composition of `run1` and `run2` is stored in a run with ID `ID`. Sequential composition means, that from each event in `run1` to each event in `run2` an LPO-edge is drawn.
- If `run1` and `run2` are runs, then by `'compose ID run1 run2'` the parallel composition of `run1` and `run2` is stored in a run with ID `ID`. Parallel composition means, that between event in `run1` and `run2` there are no edges.
- If `run` is a run, then by `'iterate ID run N'` the `run` is `N` times sequentially composed with itself (iterated) and the result is stored in a run with ID `ID`.

Using these operations, longer runs can be construced such as LPO `lpo2` shown in Figure 7. Figure 9 shows the syntax for specifying `lpo2`.

It is also possible to apply sequential composition and iteration only partially w.r.t. a so called interface. An interface specifies explicitly, which events of the

```
set set1
lpo a
event a a
opl
lpo b
event b b
opl
iterate lpo1 b 3
compose lpo2 a lpo1
tes
```

Fig. 9. Syntax for specifying LPO `lpo2`.

previous run are in direct causal dependency with which events of the subsequent run. Only between such events an edge is drawn. An interface is specified as an option of the operations `append` and `iterate` of the form `'-interface EDGELIST'`, where an edge in `EDGELIST` between events with IDs `e1` and `e2` is specified by `'e1 < e2'` and edges are separated by a space. An interface can be used to specify LPO `lpo3` shown in Figure 7. Figure 10 shows the syntax for specifying `lpo3`.

```
set set1
lpo a
event a a
opl
lpo b
event b b
opl
lpo c
event c c
opl
compose lpo1 b c
append lpo2 a lpo1
iterate lpo3 lpo2 2 -interface b<a
tes
```

Fig. 10. Syntax for specifying LPO `lpo3`.

It is possible to use a run specified in a certain set within another set by using its fully qualified ID. The same holds for events.

A run or a set of runs with ID `ID` can be stored by `'save ID FILE'` at location `FILE`. A run or set of runs stored at location `FILE` can be be loaded by `'load FILE'`. A run can be loaded only within an opened set of runs.

At each stage of the input, by `'state all'` all objects constructed so far are printed in form of text. The notation used here is the same as in the case of saving objects. If the GUI is used, by `'plot ID'` the run with ID `ID` is visualized.

A synthesis algorithm `ALG` can be applied to a set of runs with ID `ID` by `'ALG ID [OPTIONS]'`. The synthesized Petri net is stored in PNML format, such that it can be visualized by Petri net editors. If the `GUI` is used instead of the `CLI`, the Petri net is also visualized. The user is noticed, if the synthesized net is an exact solution or not. If the algorithms uses the separation representation and the net is not an exact solution, all wrong continuations which could not be prohibited are returned as a tuple $(prefix, step)$, where $prefix$ and $step$ are given by their Parikh-vector (counting the number of transition occurrences in the prefix and in the follower step). As already mentioned, only the algorithm `syn-tf-sep` is available in the download version. This algorithm has no options, so far.

The program is exited by `'exit'`.

4.5 Storing Petri nets and sets of runs

Synthesized Petri nets are stored in the Petri Net Markup Language (PNML) [20], version 2009. Runs are stored in a simple text format listing events and edges. As an example, Figure 11 shows the text file storing LPO `lpo3` from Figure 7.

```
1  lpo lpo3
2  event a a
3  event b b
4  event c c
5  event a_1 a
6  event b_1 b
7  event c_1 c
8  < a b
9  < a c
10 < a_1 b_1
11 < a_1 c_1
12 < b a_1
13 < a a_1
14 < a b_1
15 < a c_1
16 < b b_1
17 < b c_1
18 opl
```

Fig. 11. Text format for storing runs.

5 Case Studies

We tested the algorithm `syn-tf-sep` w.r.t. two aspects: Performance, and compactness and simplicity of the synthesized net.

For testing compactness, we constructed several simple Petri nets with different initial markings having a finite set of runs, synthesized a net from this set of runs and compared the result with the initial net. Figure 12 shows two of the considered Petri nets with parametrized initial marking allowing different numbers of iterations. The complete set of considered sets of runs can be downloaded with the tool. In all cases the synthesized net and the initial net coincided.

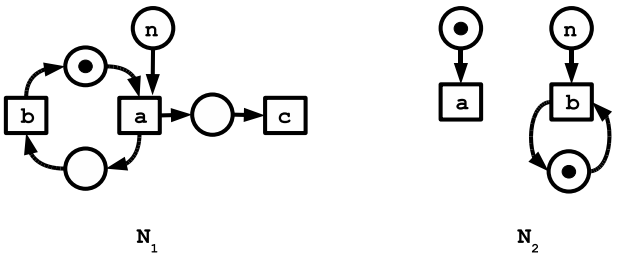

Fig. 12. Petri nets having runs `lpo1` (N_1 with $n = 1$), `lpo2` (N_2 with $n = 3$) and `lpo3` (N_1 with $n = 2$).

For testing performance we considered the following examples used in [2] for comparing performance and number of places of the synthesized net of two algorithms implemented in VIPTOOL (which also can be downloaded with the tool):

- LPOs for testing performance in presence of non-determinism (all LPOs are given in the form of step sequences): $lpo_1 = b$, $lpo_2 = a(a + b)$, $lpo_3 = c(2a)$, $lpo_4 = cb$ and $lpo_5 = cc$.
- LPOs for testing performance in presence of concurrency (the notion uses iteration of events of the form a^n and a parallel composition operator $\|$): $lpo_{6,n} = a^n \parallel b^n \parallel c^n$.

Algorithm `basis` computes place/transition nets from finite sets of LPOs using the basis representation of the set of token flow regions. Algorithm `classic` computes place/transition nets from finite sets of LPOs using the separation representation of the set of transition regions of the step language corresponding to the set of LPOs. It turned out in [2] that algorithm `basis` performed much better in case of much concurrency and little nondeterminism (test series $lpo_{6,n}$) and the other way round that algorithm `classic` performed much better in case of little concurrency and much nondeterminism (test series with combinations of lpo_1 - lpo_5). Moreover, algorithm `classic` computed smaller nets.

Our experimental results show, that algorithm `syn-tf-sep` computes as small nets as algorithm `classic`, since it also uses the separation representation. Concerning performance on the other side, algorithm `syn-tf-sep` performes much better than algorithm `classic` and little worse than algorithm `basis` for the test series $lpo_{6,n}$ (for example runtimes 13 ms for the LPO-set $\{lpo_{6,2}\}$ and 132 ms for $\{lpo_{6,3}\}$). Concerning the test series with combinations of lpo_1 - lpo_5,

it performs as fast as algorithm `classic` (for example runtimes 5 ms for the LPO-set $\{lpo_1, lpo_2\}$ and 6 ms for $\{lpo_1, lpo_2, lpo_3, lpo_4, lpo_5\}$). Altogether, it is able to cope with nondeterminism and concurrency (since we ran the algorithms `basis` and `classic` several years ago on another system at another institut as `syn-tf-sep`, it does not make sense to compare absolute runtimes).

Currently we are working on a more efficient implementation concerning concurrency. In particular, it is possible to significantly reduce the number of prefixes, which need to be computed, by considering a more compact representation of iterations (which is currently implemented in the context of infinite iterations, see Section 7).

6 Comparison to other Tools

Up to our best knowledge, the only tool which also supports synthesis from partial languages is the graphical Petri net editor VIPTOOL [11]. In VIPTOOL many synthesis algorithms for languages of LPOs of the first phase 2008 - 2010 of the project SYNOPS are implemented [5,2,3,16,15]. VIPTOOL concentrates on business process modelling and has also verification and simulation capabilities. VIPTOOL currently is further developed and maintained at Distance University in Hagen (Germany), while the project and tool SYNOPS is developed at Augsburg University (Germany). In contrast to VIPTOOL, the SYNOPS tool supports more kinds of causal structures and Petri net classes and more general classes of infinite partial languages (see section 7). It only concentrates on synthesis capabilities and is text based. It mainly serves for rapid implementation and evaluation of newly developed term based representations of infinite partial languages and synthesis algorithms. For such term based representation and synthesis algorithms, which turn out to be stable, an integration into VIPTOOL is planned.

There is is another tool-supported line of research considering transition systems instead of languages as behavioral specification. The tool [6] computes distributable bounded Petri nets from such specifications. In [8,7] tools are described which synthesize labelled Petri nets with non-unique transition names (here the techniques are different to the presented ones).

One application of synthesis algorithms is process mining. There is a big tool frame work called PROM [17] which integrates many different mining and analysis capabilities concerning process modells and event logs. The mining tools are based on descriptions of the sequential behavior of systems (which cannot directly represent concurrency).

7 Outlook

Currently, an analoguous algorithm is implemented for the synthesis of general inhibitor nets from finite sets of LSOs [21] based on results in [15]. For this, some new ideas concerning wrong continuation were developed (which are not presented here due to lack of space). Within another bachelor thesis, operations

for the specification of infinite sets of LPOs and a corresponding synthesis algorithm are implemented at the time of writing [18] based on results in [5,14]. In [19] a synthesis algorithm which computes place/transition nets from finite sets of LDAGs is described. This algorithm still needs some improvements which are currently implemented.

The presented set of operations is currently extended by the following operations allowing fast generation of large sets of runs: Alternative composition of runs, sequential composition, parallel composition and iteration of sets of runs, and standard operations on sets (of runs) like union, intersection, difference.

In order to increase usability, we plan to implement shortcuts for all operations (such a `'a<b'` instead of `'et a b'` or `'lpo1 a<b'` instead of `'append lpo1 a b'`) and the possibility to use more than one operation in a command (such as `'lpo1 a<(b|c)'` instead of the sequence `'compose lpo0 b c'` and `'append lpo1 a lpo0'`).

Further steps are: Adapting the algorithms to restriced net classes such as elementary nets and workflow nets and to the use of additional information such as predefined places or undesired runs.

8 Download

The tool can be downloaded from the project webpage [13]. There are executable program files for 32 Bit and 64 Bit Windows systems, with and without GUI. On the webpage you also find the example sets of runs we used to evaluate the tool.

References

1. R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Process Mining Based on Regions of Languages. In G. Alonso, P. Dadam, and M. Rosemann, editors, *BPM*, volume 4714 of *Lecture Notes in Computer Science*, pages 375–383. Springer, 2007.
2. R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Synthesis of Petri Nets from Finite Partial Languages. *Fundam. Inform.*, 88(4):437–468, 2008.
3. R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Synthesis of Petri Nets from Scenarios with Viptool. In K. M. van Hee and R. Valk, editors, *Petri Nets*, volume 5062 of *Lecture Notes in Computer Science*, pages 388–398. Springer, 2008.
4. R. Bergenthum, J. Desel, S. Mauser, and R. Lorenz. Construction of Process Models from Example Runs. *T. Petri Nets and Other Models of Concurrency*, 2:243–259, 2009.
5. R. Bergenthum, J. Desel, S. Mauser, and R. Lorenz. Synthesis of Petri Nets from Term Based Representations of Infinite Partial Languages. *Fundam. Inform.*, 95(1):187–217, 2009.
6. B. Caillaud. Synops-Homepage., 2002. http://www.informatik.uni-augsburg.de/lehrstuehle/ inf/projekte/synops/.
7. J. Carmona, J. Cortadella, and M. Kishinevsky. Genet: A Tool for the Synthesis and Mining of Petri Nets. In *ACSD*, pages 181–185. IEEE Computer Society, 2009.

8. J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers. *IEICE Trans. of Informations and Systems*, E80-D(3):315–325, 1997.
9. J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Hardware and Petri Nets: Application to Asynchronous Circuit Design. In *ICATPN 2000, LNCS 1825*, pages 1–15. Springer, 2000.
10. J. Desel. From Human Knowledge to Process Models. In R. Kaschek, C. Kop, C. Steinberger, and G. Fliedl, editors, *UNISCON*, volume 5 of *Lecture Notes in Business Information Processing*, pages 84–95. Springer, 2008.
11. J. Desel. VipTool-Homepage., 2010. http://www.fernuni-hagen.de/se/viptool.html.
12. M. B. Josephs and D. P. Furey. A Programming Approach to the Design of Asynchronous Logic Blocks. In *Concurrency and Hardware Design 2002, LNCS 2549*, pages 34–60. Springer, 2002.
13. R. Lorenz. Synops-Homepage., 2010. http://www.informatik.uni-augsburg.de/lehrstuehle/ inf/projekte/synops/.
14. R. Lorenz, J. Desel, and G. Juhas. Models from scenarios. In *Proccedings of "Advanced Course on Petri Nets 2010", T. Petri Nets and Other Models of Concurrency*, Lecture Notes in Computer Science. Springer, to appear in 2012.
15. R. Lorenz, S. Mauser, and R. Bergenthum. Theory of Regions for the Synthesis of Inhibitor Nets from Scenarios. In J. Kleijn and A. Yakovlev, editors, *ICATPN*, volume 4546 of *Lecture Notes in Computer Science*, pages 342–361. Springer, 2007.
16. R. Lorenz, S. Mauser, and G. Juhás. How to synthesize Nets from Languages: a Survey. In S. G. Henderson, B. Biller, M.-H. Hsieh, J. Shortle, J. D. Tew, and R. R. Barton, editors, *Winter Simulation Conference*, pages 637–647. WSC, 2007.
17. Process Mining Group Eindhoven Technical University: ProM-Homepage. http://www.promtools.org/prom5/.
18. J. Robl. Synthese von Petrinetzen aus unendlichen Mengen beschrifteter partieller Ordnungen, to be finished july 2012. Bachelor thesis, Universität Augsburg.
19. K. Rüschenbaum. Synthese von Petrinetzen aus endlichen Mengen beschrifteter, gerichteter, azyklischer Graphen, 2011. Bachelor thesis, Universität Augsburg.
20. P. team. PNML.org: The Petri Net Markup Language home page, 8 2011. `http://www.pnml.org/`.
21. M. Urban. Synthese von Inhibitornetzen aus endlichen Mengen beschrifteter, geschichteter Ordnungen, to be finished march 2012. Bachelor thesis, Universität Augsburg.
22. W. M. P. van der Aalst and C. W. Günther. Finding Structure in Unstructured Processes: The Case for Process Mining. In *ACSD*, pages 3–12. IEEE Computer Society, 2007.
23. W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. Workflow Mining: A Survey of Issues and Approaches. *Data Knowl. Eng.*, 47(2):237–267, 2003.
24. S. Warshall. A Theorem on Boolean Matrices. *Journal of the ACM 9*, (1):11–12, 1962.
25. M. Zhou and F. D. Cesare. *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems*. Kluwer, 1993.