Experimental Results on the Synthesis of Petri Nets from Partial Languages*

Robin Bergenthum, Sebastian Mauser

Lehrstuhl für Angewandte Informatik
Katholische Universität Eichstätt-Ingolstadt, Eichstätt, Germany
e-mail: {robin.bergenthum, sebastian.mauser}@ku-eichstaett.de

Abstract

In [8] a synthesis algorithm for place/transition-nets from finite partial languages was developed. In this paper we present an implementation into VipTool [5] and experimental results for this synthesis algorithm.

1 Introduction

Synthesis of Petri nets from behavioural 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 system design, and recently also in workflow design.
Originally, synthesis means algorithmic *construction of a Petri net* from sequential observations. It can be applied to various classes of Petri nets, including elementary nets [6, 7] and place/transition nets (p/t-nets) [1]. Synthesis can start with a transition system (*synthesis up to isomorphism*) representing the sequential behaviour of a system or with a step transition system which additionally represents steps of concurrent events [1]. Synthesis can also be based on a language (*synthesis up to language equivalence*), e.g. a set of occurrence sequences or step sequences [4]. The classical *synthesis problem* is the problem to decide whether, for a given behavioural specification (transition system, language), there exists an unlabelled Petri net of the respective class, such that the behaviour of this net coincides with the specified behaviour. In the positive case usually a solution net is constructed.
Recently, we solved the synthesis problem for p/t-nets with behaviour given in terms of a *finite partial language*, i.e., as a finite set of labelled partial orders (LPOs) [8].[1] In contrast to previous work on the synthesis problem, we considered partial order behaviour of Petri nets, truly representing the concurrency of events. Partial orders are often considered the most appropriate representation of behaviour of concurrent systems modelled by Petri nets. The *decision algorithm* presented in [8] for the synthesis problem in the considered setting (behavioural specification: partial language, target Petri net class: p/t-nets) is based on the so-called *theory of regions*. We now implemented the algorithm and integrated it in a beta version of our framework VipTool [5]. In this paper we will explain some algorithmic details of the *implementation* of this synthesis algorithm (going beyond the scope of the paper [8]) and finally present *experimental results* (performance tests, etc.).
The synthesis algorithm is divided into *two parts*. First a p/t-net having the smallest partial order behaviour including the specified partial language is constructed. This net represents a partial language being a best upper approximation of the given partial language through net behaviour. Thus either this net

*Supported by the German research council - project "Synthesis of Petri nets from Scenarios" (SYNOPS)

[1] LPOs are also known as partial words or pomsets.

solves the synthesis problem positively or the synthesis problem has a negative answer. This is checked by the second part of the decision algorithm, i.e. it is checked whether the partial order behaviour of the constructed net coincides with the specified partial language. For this second part two alternative procedures are proposed in [8], the so-called "optimistic" and the so-called "pessimistic equality test". In this paper we only consider the "optimistic equality test". For practical applications in particular the first part of the synthesis algorithm is of interest, because the main focus usually lies in the construction of a system model from a given specification (not in the decision of the synthesis problem). In this context the first algorithm part is a useful standalone algorithm to compute a p/t-net system, which is a best upper approximation for a set of scenarios specified in terms of LPOs. The central idea of the computation of the p/t-net is sketched in the cover picture: While the transitions of the synthesized net are given by the labels of the LPOs, the places are given by the rays of a pointed polyhedral cone, which is defined by an inequation system having the set of edges of the LPOs as variables (details are explained in the next section).

2 The Synthesis Algorithm

In this section the two parts of the synthesis algorithm, in particular important implementation details, are explained with a running example. Figure 1 shows a set of two LPOs lpo1 and lpo2 designed in VipTool. This set represents the specified partial language in the running example.

Place/transition-nets (p/t-nets) are general Petri nets regarding arc weights (e.g. see the cover picture or the net basisNet in Figure 1). The labels of events of a given LPO refer to transition occurrences in a p/t-net. The order relation of an LPO is interpreted as an "earlier than" relation between transition occurrences. Unordered events are concurrent. A given LPO is an execution, i.e. included in the partial order behaviour, of a p/t-net, if the events of the LPO can occur in the p/t-net respecting the order relation and the concurrency relation of the LPO. For example the second LPO lpo2 in Figure 1 is an execution of a p/t-net, if and only if in the initial marking of the p/t-net, the transition A is enabled, and after the occurrence of A, the transitions A and B are concurrently enabled. That means lpo2 is enabled in the p/t-net depicted on the cover picture as well as in the p/t-net basisNet in Figure 1.

2.1 First Part – Computing a Best Upper Approximation

The first part of our synthesis algorithm computes a p/t-net, being a best upper approximation to a p/t-net having the specified partial order behaviour, as follows: The transitions of the p/t-net are given by the finite set of labels of the partial language, i.e. A and B in the running example. Adding places to the p/t-net restricts the enabledness of these transitions. Places are defined by their initial marking and the weights on the arcs connecting them to each transition. Adding only places to the p/t-net, which do not prohibit any specified LPO from being an execution of the p/t-net, guarantees the partial order behaviour of the computed p/t-net to be an upper approximation of the specified partial language. Such places are called feasible. Adding all feasible places obviously generates a best upper approximation, the so called saturated feasible p/t-net. This net generates the specified behaviour and minimal additional behaviour, but it has infinitely many places. So-called regions enable the computation of a finite subset of the set of all feasible places generating the same behaviour. Such set of places is finally added to the synthesized p/t-net.

Regions of a partial language define the set of all feasible places structurally on the level of the given partial language. The idea of defining regions is as follows: If two events are ordered in an LPO this specifies that the corresponding transition occurrences may be causally dependent. Such a causal dependency arises exactly if the first transition occurrence produces tokens in a place, and some of these tokens are consumed by the second transition occurrence. Such a place can be defined as follows: Assign to every edge of an LPO a non-negative integer, a so-called token flow, representing the number of tokens which

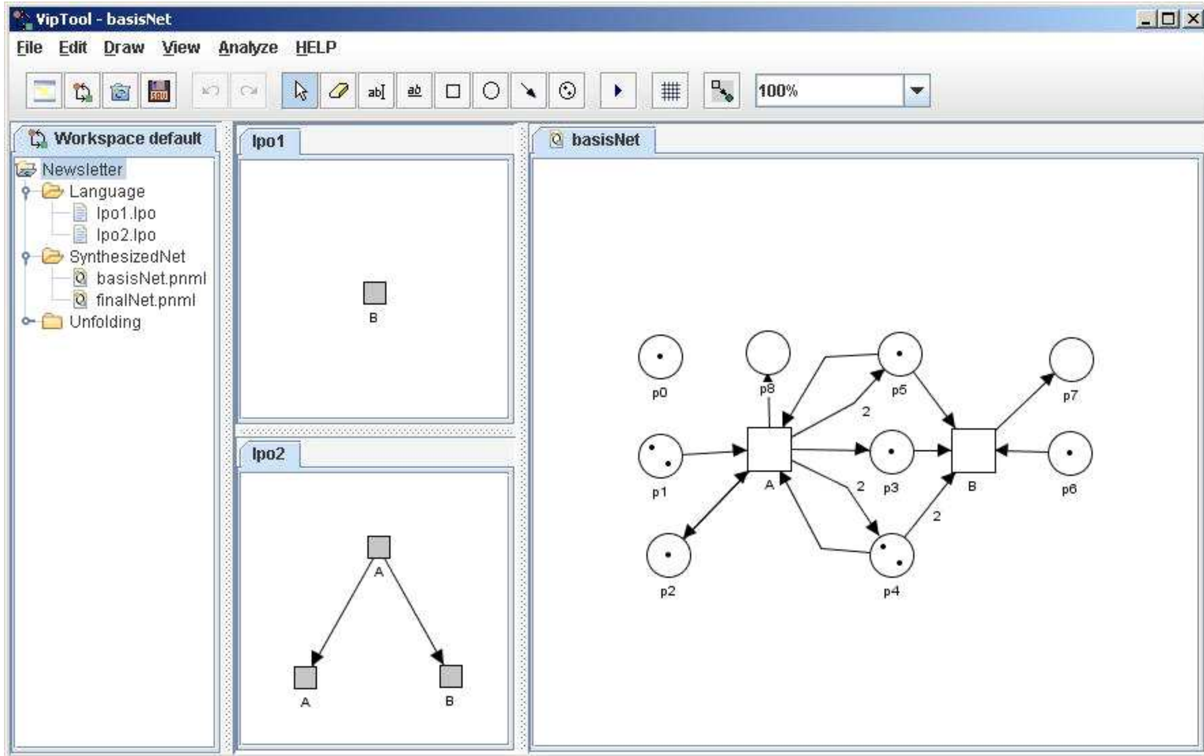

Figure 1: A screenshot of VipTool: On the left side it shows two LPOs (lpo1 and lpo2) drawn in the VipTool editor. These two LPOs form the specified partial language in our running example. On the right side the p/t-net (basisNet) consisting of all places defined by basis regions of the partial language is depicted.

are produced by the first transition occurrence and consumed by the second transition occurrence in the place to be defined. Then the number of tokens consumed overall by a transition occurrence in this place is given as the sum of the token flows assigned to ingoing edges of the event, the so-called intoken flow. This number can then be interpreted as the weight of the arc connecting the new place with the respective transition. Similarly, the number of tokens produced overall by a transition occurrence in this place is given as the sum of the token flows assigned to outgoing edges of the event, the so-called outtoken flow. This number can then be interpreted as the weight of the arc connecting the respective transition with the new place. Moreover, transition occurrences can also consume tokens from the initial marking of the new place (tokens which are not produced by another transition occurrence): In order to specify the number of such tokens, we extend each LPO by an initial event labelled by the same symbol (S in the cover picture) representing a start-transition producing the initial marking. The initial event is in "earlier than" relationship to all other events of the LPO. The outtoken flow of this event can be interpreted as the initial marking of the new place. Similarly transition occurrences can produce tokens in the new place which remain in the final marking after the execution of the LPO (tokens which are not consumed by some subsequent transition occurrence): In order to specify the number of such tokens, we extend each LPO by a final event labelled by different symbols (E and F in the cover picture) representing tran-

sitions consuming the final marking. All the other events of an LPO are in "earlier than" relationship to the final event. The LPOs of the running example extended by initial and final events are depicted on the cover picture. The edges of all LPOs are enumerated. These unique numbers are annotated in circles as "identifiers" to the edges. We refer to the token flow on a certain edge by these numbers in circles.

In the last paragraph we explained how to define a place by adding non-negative integers to each edge of the set of extended specified LPOs. The weight on the arc ingoing to (outgoing from) the place and outgoing from (ingoing to) a certain transition is defined by the outtoken (intoken) flow of some event (of some specified LPO) labelled by the respective transition. The initial marking of the place is defined by the outtoken flow of an initial event of some LPO. This is only well-defined if the outtoken and intoken flows of equally labelled events (including the initial events) in the partial language coincide. A mapping adding token flows to the edges of the set of extended specified LPOs, which fulfils this property, is called token flow function. Each token flow function defines a place. An example token flow function for the running example is given by $(\textcircled{1},\textcircled{2},\textcircled{3},\textcircled{4},\textcircled{5},\textcircled{6},\textcircled{7},\textcircled{8},\textcircled{9},\textcircled{10},\textcircled{11},\textcircled{12}) = (0,1,0,0,1,0,0,1,1,0,0,2)$ (the numbers in circles refer to the token flows on the respective edges, see the cover picture). The initial marking of the place defined by this token flow function is one and the weights on the edges to (from) A and B are one and one (two and zero). A weight of zero means that there is no edge. The example place can be found on the cover picture as well as in the p/t-net basisNet in Figure 1. In [8] we showed that the set of places defined by a token flow function coincides with the set of feasible places. A token flow function is called region of the given partial language.

Altogether a region of a finite partial language is simply a finite vector assigning token flows, i.e. non-negative integers, to the edges of the set of extended specified LPOs, such that the intoken and outtoken flows of equally labelled nodes coincide. The latter condition can easily be encoded in homogeneous linear equations. In the running example we need an equation (i) to ensure that the two S-labelled (initial) events have equal intoken flow, an equation (ii) to ensure that the two A-labelled events have equal intoken flow, an equation (iii) to ensure that the two A-labelled events have equal outtoken flow, an equation (iv) to ensure that the two B-labelled events have equal intoken flow and an equation (v) to ensure that the two B-labelled events have equal outtoken flow:

$$(i) \quad \textcircled{1} + \textcircled{2} - \textcircled{4} - \textcircled{5} - \textcircled{6} - \textcircled{7} = 0$$

$$(ii) \quad \textcircled{5} - \textcircled{7} - \textcircled{9} = 0$$

$$(iii) \quad \textcircled{8} + \textcircled{9} + \textcircled{10} - \textcircled{12} = 0$$

$$(iv) \quad \textcircled{2} - \textcircled{6} - \textcircled{8} = 0$$

$$(v) \quad \textcircled{3} - \textcircled{11} = 0$$

The non-negativity condition of regions defines one homogeneous inequation for each variable, i.e. for each edge. In the running example we have $\textcircled{1} \geq 0, \textcircled{2} \geq 0, \textcircled{3} \geq 0, \textcircled{4} \geq 0, \textcircled{5} \geq 0, \textcircled{6} \geq 0, \textcircled{7} \geq 0, \textcircled{8} \geq 0, \textcircled{9} \geq 0, \textcircled{10} \geq 0, \textcircled{11} \geq 0, \textcircled{12} \geq 0$. Finally only integers are allowed as entries of a region vector. Altogether the set of regions equals the set of non-negative integer solutions of a homogeneous linear equation system with integer coefficients. The number of equations is linear in the number of nodes of the specified partial language. The number of variables equals the number of edges of the set of extended specified LPOs. The discussed equation system together with inequations ensuring the non-negativity of the variables (as shown above), defines a pointed polyhedral cone. Such a cone is illustrated on the cover picture (in three instead of twelve dimensions). The set of regions is the set of integer points of this cone. A pointed polyhedral cone is generated by the finite set of its rays. More precisely choosing one vector on each ray results in a minimal basis of the cone. That means every vector of the cone can be generated as a non-negative linear combination of this basis and the basis is minimal (in the number of elements) with this property. Since in our case the considered equations have integer coefficients, this basis can be chosen as integer vectors, i.e. as regions. We showed in [8] that the finite net having the set of places defined by this set of basis regions has the same partial order

behaviour as the infinite net consisting of all feasible places. In the running example this finite net is the p/t-net basisNet depicted in Figure 1. The correspondence of the computed places to the rays of the considered cone is illustrated on the cover picture by drawing places onto the rays of the depicted cone.

The computed set of places defined by basis regions still contains many implicit places, which can be deleted without changing the behaviour of the synthesized net. Therefore a simple and efficient procedure to delete implicit places of the synthesized net is included in the first part of our synthesis algorithm. This procedure deletes places, that are dominated (w.r.t. the behavioural restriction) by one another place, but combinations of places dominating other places are not searched. Applying this procedure, the algorithm returns the net finalNet depicted in Figure 2 (this net is also shown on the cover picture), i.e. the first part of our synthesis algorithm computes this net. The places p_0 , p_3 , p_7 and p_8 are deleted from the p/t-net basisNet of Figure 1, since p_0 , p_7 and p_8 induce no behavioural restriction at all and p_3 is dominated by p_6 . Advanced methods to detect implicit places (considering combinations of places) still offer extensive improvement possibilities for this module of our synthesis algorithm.

The implementation of the first part of the synthesis algorithm is as follows: First the equation system defining the cone is constructed from the given partial language as shown before. The integer basis of the cone is computed from the equation system using the algorithm of Tschernikow [3]. This algorithm starts by constructing a matrix consisting of the identity matrix and the transpose of the matrix defining the considered equations horizontally arranged. The matrix is stepwise transformed. In each step a non-zero lead column corresponding to one of the equations is chosen and so called balanced pairs of rows are transformed to equilibrium rows w.r.t. the lead column. After such transformation the previous lead column only contains zero values. One transformation step may increase the number of rows at most quadratically. The algorithm finishes, when each column corresponding to an equation is zero, i.e. it performs at most as many steps as the number of equations. The non-negative integer basis solutions of the considered equation system can then directly be read out from the last constructed matrix. These solutions are the basis regions. A p/t-net is then synthesized by adding a transition for each label in the partial language and a place for each computed basis region as explained before (in the running example leading to the p/t-net basisNet in Figure 1). Implicit places dominated by one another place are finally deleted by comparing the computed places pairwise. In the running example this leads to the p/t-net depicted on the cover picture. Altogether the net on the cover picture is the final synthesized net of the first part of our synthesis algorithm applied to the running example. The cone behind the net illustrates the computation principle of the places of this net.

Concerning the performance of this first part of the synthesis algorithm, the crucial factor is the Tschernikow algorithm. It computes the non-negative integer basis of an integral homogeneous equation system, which has linear size in the size of the input partial language. The algorithm is very well suited for our purposes, because it is developed for computing basis solutions for the set of non-negative solutions of a homogeneous equation system as it is given in our situation. Tschernikow exploits this setting to get an improved version of a previous algorithm to compute the basis of a homogeneous inequation systems developed by Motzkin and Burger. Since we have a sparse matrix (i.e. many entries of a lead column are already zero), the algorithm should perform quite well in our setting. In particular the growth of the number of rows of the stepwise computed matrices should be very limited, often the number even decreases in a transformation step. Nevertheless the runtime may be exponential in worse cases. Also the size of the finally computed basis may be exponential in the worst case. But usually it has a reasonable size. Moreover some places defined by basis regions are deleted as implicit places in the final synthesized net. The final number of places is important, because the synthesized net is input for the second part of the synthesis algorithm. Moreover the constructed net may be of interest for further analysis. Thus a reasonable size of this net is especially desired. The performance of this first part of our synthesis algorithm, and in particular the size of the synthesized net, is tested with

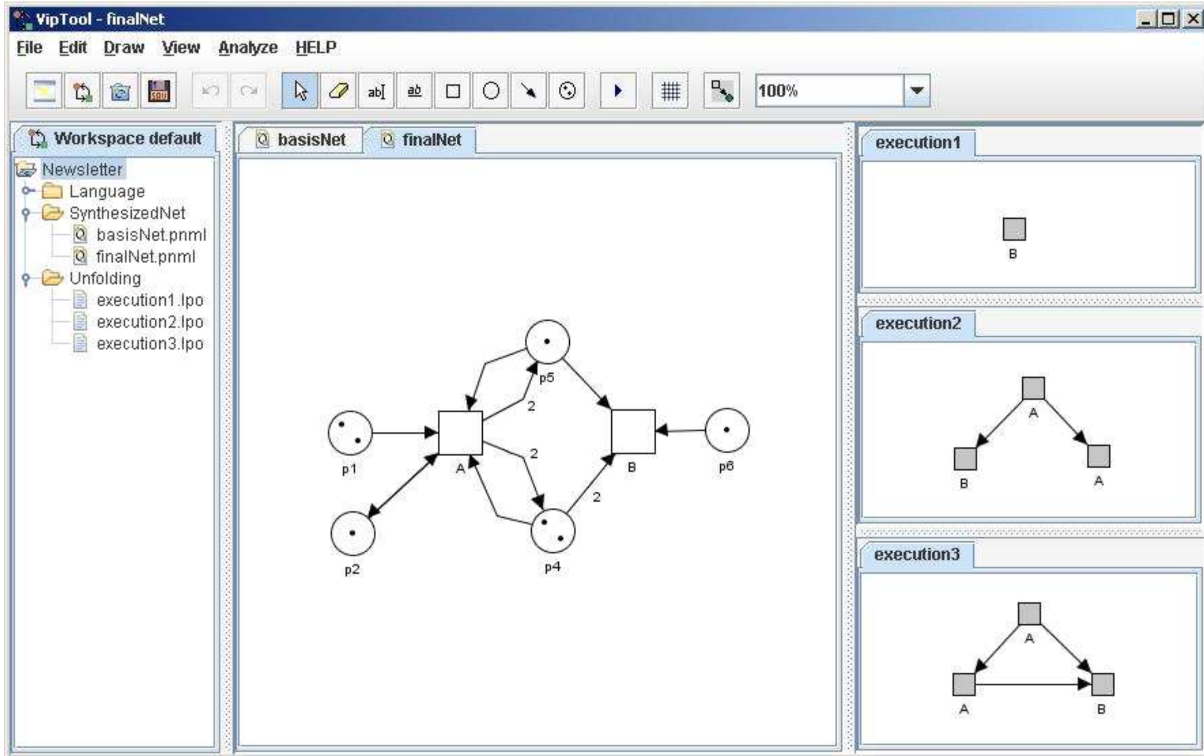

Figure 2: A screenshot of VipTool: On the left side the p/t-net (finalNet) resulting from the p/t-net basisNet (Figure 1) by deleting places, dominated by one another place, is depicted. The net is also depicted on the cover picture. This net is synthesized by the first part of the synthesis algorithm applied to the partial language of the running example (Figure 1). On the right side the executions of the p/t-net finalNet (execution1, execution2 and execution3) computed by our unfolding algorithm are shown.

practical examples in the next section.

2.2 Second Part – Equality Test

The second part of the synthesis algorithm – the optimistic equality test, whether the computed net has exactly the specified partial order behaviour – consists of two steps: In a first step the partial order behaviour of the constructed net is computed, and in the second step it is tested, if this behaviour is specified in the given partial language. Note here, that from the best upper approximation property of the synthesized p/t-net, we already know the reverse, that the specified partial language is included in the partial order behaviour of the p/t-net. The partial order behaviour of the p/t-net is computed by an unfolding algorithm. We implemented a newly developed very fast unfolding algorithm to compute the partial order behaviour of p/t-nets. In [2] we present this algorithm and demonstrate its superior performance in contrast to classical unfolding algorithms. The algorithm constructs all partial orders corresponding to finite Goltz-Reisig processes of maximal length of the p/t-net. We showed in [8] that the synthesized p/t-net has only finite Goltz-Reisig processes. Thus the computed set of partial orders is a set of executions of maximal length of the p/t-net including all minimal (w.r.t. the causal orderings) ex-

ecutions of the net. In the running example the synthesized p/t-net, that has to be unfolded, is the net finalNet shown in Figure 2. Figure 2 shows the set of executions (execution1, execution2 and execution3), of this net computed by our unfolding algorithm. In a second step it is checked if each computed execution of the synthesized p/t-net is specified. That means for each such execution it is tested, if it is isomorphic to a sequentialization (i.e. having a bigger causality relation) of one of the specified LPOs. Since the computed executions include all minimal executions of maximal length, this check actually shows, if the partial order behaviour of the p/t-net is specified in the partial language. Thus a positive check implies a positive answer to the synthesis problem having the synthesized p/t-net as a witness, and a negative check implies a negative answer. In the running example it is obvious, that the computed set of executions of the synthesized p/t-net (Figure 2) is specified in the given partial language (Figure 1), since execution1 equals lpo1, execution2 equals lpo2 and execution3 sequentializes lpo2. Thus we have a positive answer to the synthesis problem in this case and the net on the cover picture is a solution net.

Concerning the performance of this second part of the synthesis algorithm, one has to regard that in general, the number of Goltz-Reisig processes of a p/t-net is exponential in the size of the p/t-net. This is problematic, because each computed execution has to be tested in the second step of this part of the synthesis algorithm. But in our special situation we expect that the number of computed executions roughly coincides with the size of the given partial language, because the computed executions form a best upper approximation to the given language. A further problem is that the calculation of this set of executions requires an exponential runtime in the worst case, but our unfolding algorithm is at least very efficient compared to classical unfolding algorithms. The test, if a computed execution is isomorphic to a sequentialization of one of the specified LPOs, is a special graph isomorphism problem. Graph isomorphism problems are assumed to form an own complexity class between P and NP. The common procedure to solve graph isomorphism problems is applying backtracking algorithms constructing a set of possible isomorphism, and then checking each possible isomorphism, if it actually is an isomorphism. The efficiency of the procedure depends on the quality of restricting the set of possible isomorphisms in the backtracking algorithm. Our implementation uses such a backtracking procedure. It applies a very restrictive process of eliminating possible isomorphisms by identifying certain equivalence classes of events. These equivalence classes account for the label of the event as well as the labels of all events in the pre- and post-set of the event. This is very restrictive because of the transitivity of LPOs. Thus an efficient isomorphism test is ensured. The experimental results in the next section reveal the performance of the two steps of the second part of the synthesis algorithm in practice.

We implemented the synthesis algorithm for p/t-nets from partial languages as described in this section into our framework VipTool [5]. Figure 1 and Figure 2 show examples of the user interface of VipTool. The partial language may be specified in the VipTool editor or as an xml-file. The synthesized net is stored as a pnml-file and can be visualized with VipTool.

3 Experimental Results

In this section we show experimental results of the implementation of the described synthesis algorithm.[2] The considered partial languages are subsets of the LPOs shown in Figure 1, Figure 3 and Figure 4. The table on the next page shows the runtime in milli seconds ("ms") of the first part of the synthesis algorithm ("Synthesis") and the runtime of the second part (optimistic equality test) divided into the unfolding procedure ("Unfolding") and the comparison of the computed executions and the specified partial language ("Test"). We also show the number of basis regions ("basis") and the final number of places ("places") of the net constructed in the first part of the synthesis algorithm as well as the final result of the equality test ("result").

Concerning the runtime in particular the unfolding algorithm seems problematic (the runtime entry "-"

[2] We used Java SE 1.5.0 on an Intel Xeon 2.8 GHz machine with 3072 MB RAM running Microsoft Windows Server 2003 SE operating system.

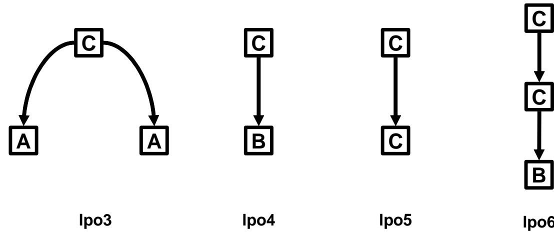

Figure 3: Example LPOs lpo3, lpo4, lpo5, lpo6.

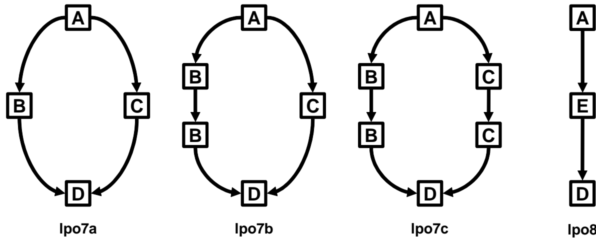

Figure 4: Example LPOs lpo7a, lpo7b, lpo7c, lpo8.

means that the algorithm did not finish in reasonable time), although we have already chosen an efficient unfolder. We are currently working on an even more efficient unfolding algorithm. Also further methods to delete implicit places in the constructed net could improve the situation. Lastly it could also be interesting to implement the so-called pessimistic equality test, which does not include an unfolding algorithm, instead of the optimistic test used here.

References

[1] E. Badouel and P. Darondeau. On the synthesis of general petri nets. Technical Report 3025, Inria, 1996.

[2] R. Bergenthum, R. Lorenz, and S. Mauser. Faster unfolding of general petri nets. In *14. Workshop Algorithmen und Werkzeuge für Petri Netze (AWPN)*, 2007.

[3] S. N. Cernikov. *Lineare Ungleichungen.* Deutscher Verlag der Wissenschaften, VEB, 1971.

[4] P. Darondeau. Deriving unbounded petri nets from formal languages. In D. Sangiorgi; R. de Simone, editor, *CONCUR*, volume 1466 of *Lecture Notes in Computer Science*, pages 533–548. Springer, 1998.

[5] J. Desel, G. Juhás, and R. Lorenz. Viptool-homepage., 2003. http://www.informatik.ku-eichstaett.de/projekte/vip/.

[6] A. Ehrenfeucht and G. Rozenberg. Partial (set) 2-structures. part i: Basic notions and the representation problem. *Acta Inf.*, 27(4):315–342, 1989.

[7] A. Ehrenfeucht and G. Rozenberg. Partial (set) 2-structures. part ii: State spaces of concurrent systems. *Acta Inf.*, 27(4):343–368, 1989.

[8] R. Lorenz, R. Bergenthum, S. Mauser, and J. Desel. Synthesis of petri nets from finite partial languages. In *Proceedings of the 7th International Conference on Application of Concurrency to System Design (ACSD)*, pages 157–166, 2007.

Language	**Synthesis**			**Unfolding**	**Test**	
	ms	basis	places	ms	ms	result
{*lpo1, lpo2*}	131	9	5	62	5	true
{*lpo1, lpo2, lpo3*}	266	35	7	73	8	true
{*lpo1, lpo2, lpo3, lpo4*}	390	88	12	110	24	true
{*lpo1, lpo2, lpo3, lpo4, lpo5*}	1.890	220	17	-	-	-
{*lpo1, lpo2, lpo6*}	359	48	14	5.718	183	false
{*lpo7a, lpo8*}	279	49	19	93	3	true
{*lpo7b, lpo8*}	407	149	27	1188	113	true
{*lpo7c, lpo8*}	734	321	41	-	-	-