Analysis of Petri Nets and Transition Systems

Eike Best and Uli Schlachter*
Department of Computing Science, Carl von Ossietzky Universität
D-26111 Oldenburg, Germany
{eike.best,uli.schlachter}@informatik.uni-oldenburg.de

This paper describes a stand-alone, no-frills tool supporting the analysis of (labelled) place/transition Petri nets and the synthesis of labelled transition systems into Petri nets. It is implemented as a collection of independent, dedicated algorithms which have been designed to operate modularly, portably, extensibly, and efficiently.

Keywords: Analysis, Labelled Transition Systems, Petri Nets, Synthesis.

1 Motivation

Labelled transition systems are frequently employed in order to display the state space of a given Petri net and to analyse its behavioural properties. Conversely, by region theory [1], a Petri net may be synthesisable from a given labelled transition system. Such a net is then correct "by design". However, a transition system may be extremely (even infinitely) large, causing synthesis algorithms to be prohibitively time-consuming. Moreover, synthesis suffers from nondeterminism, since for a given transition system, many different Petri net implementations may exist.

In such a context, it is interesting to discover relationships between special, albeit useful, classes of transition systems and classes of Petri nets (e.g., persistent ones [15]), so that faster and more deterministic analysis and synthesis methods can be devised. For the working mathematician, this tends to involve the error-prone examination of graphs which may be large and intricate. Tools such as `synet` [9] and `petrify` [12] are helpful, but there is also a need for multifunctional tools with the following properties:

- Versatility. The user should be able to create, modify, and manage hundreds or thousands of medium-sized graphs (both Petri nets and transition systems) which might only slightly be at variance with each other. E.g., in `synet`, the only way of inserting comments on data objects is by choosing meaningful file names. For large collections of objects, a more flexible commenting function becomes mandatory. No restrictions should be imported from intended applications. E.g., `petrify` excludes non-safe Petri nets as output because they are of no interest in a hardware context.
- Transparency. The tool's internal machinations should be detectable, if necessary by examining the source code. E.g., it is not known whether `synet` always constructs a safe Petri net if there exists one.

*The authors are supported by the German Research Foundation (DFG) project ARS (Algorithms for Reengineering and Synthesis), reference number Be 1267/15-1.

S. Knight, I. Lanese, A. Lluch Lafuente and H. T. Vieira (Eds.):
8th Interaction and Concurrency Experience (ICE 2015)
EPTCS 189, 2015, pp. 53–67, doi:10.4204/EPTCS.189.6

- Extensibility. It should be possible to program and add modules fast, in case the need arises for any particular new problems. In particular, modules should have properly defined, readable, and descriptive input/output interfaces.
- Bare-bonedness. The tool should operate on place/transition nets with arbitrary arc weights and side-conditions, and on arbitrary labelled transition systems, as well as on many interesting sub- (rather than super-) classes. Emphasis should be on algorithmic optimisation, rather than on textual expressiveness. Communication between users, as well as between tools, should be achieved via human-readable text files.
- Efficiency and modularity. Analysis of medium-sized objects (say, graphs with a few hundred nodes) should be fast, even if the theoretical complexity is PSPACE-hardness or worse. In the event of bottlenecks, the tool should be sufficiently modular so that the culprit(s) can be isolated quickly. Memory should be organised in such a way that average-sized objects can be handled and overflow does not occur, or can at least be localised cleanly.
- Portability and availability. It should be possible to switch quickly between different platforms. No frequent recompiling should occur, and any dependencies on residual installations should be minimised. The tool should be freely downloadable and usable as a single executable file on many different platforms. No registration or other "paperwork" (such as sending mails or waiting for release links), and few system-dependent installations, should be necessary in order to use it.

Since a tool of this kind was found to be lacking, a students' project was initiated at the University of Oldenburg in 2012. The toolbox that resulted from it by March 2013 has been called APT for A*nalysis of* P*etri nets and* T*ransition systems* and is available at [8]. Since then, APT has been optimised and extended by the second author (and other persons). The present paper contains a brief summary of the use and structure of APT in sections 2 and 3, respectively. Some recent developments will be described in section 4. Formal definitions can be found in section A. Many of them conform with [6, 7] where a more detailed exposition of some of the theory can be found.

2 Introduction to the use of APT, and some examples

APT is implemented in Java 7 and is released under the GPLv2 license. As one of the goals was portability, it consists of a single file called `apt.jar` which can be run by any Java 7 runtime environment. Currently there is no graphical user interface, but instead a console-based one. This decision was made to be able to focus on the implementation of algorithms. Listing 1 shows how APT can be downloaded with `git` and built with `ant`. As an alternative to using `ant`, the file `apt.jar` can simply be copied from another machine. Presently, no pre-compiled versions are available for download. Listing 1 also illustrates the use of APT's `help` function.

Figure 1 shows a labelled transition system, *lts*, and three Petri nets, N_1–N_3, serving as running examples. All three Petri nets are solutions of *lts*, that is, their reachability graphs are isomorphic to *lts*. Listing 2 represents N_1 in APT's file format. The file starts with a name and a description of the net. N_1 has five places named `p0` to `p4`, and four transitions, `a` to `d`. The flows of the net are specified in multiset notation. For example, transition `a` takes a token from place `p0` and puts it on `p4`. Weights can be specified either by mentioning a place multiple times, e.g. `{p,p}`, or by explicitly specifying a weight, as in `2*p`. The initial marking of the net is represented in a similar format. Comments can be enclosed within `/*..*/` or begin with `//` and extend to the end of the line. APT's transition-centred way of specifying place/transition nets

```
$ git clone http://github.com/CvO-theory/apt.git
$ cd apt
$ ant jar
$ java -jar apt.jar help bounded
Usage: apt bounded <pn> [<k>]
  pn          The Petri net that should be examined
  k           If given, k-boundedness is checked
Check if a Petri net is bounded or k-bounded.
```

Listing 1: Downloading and building APT. Some output is omitted for reasons of brevity.

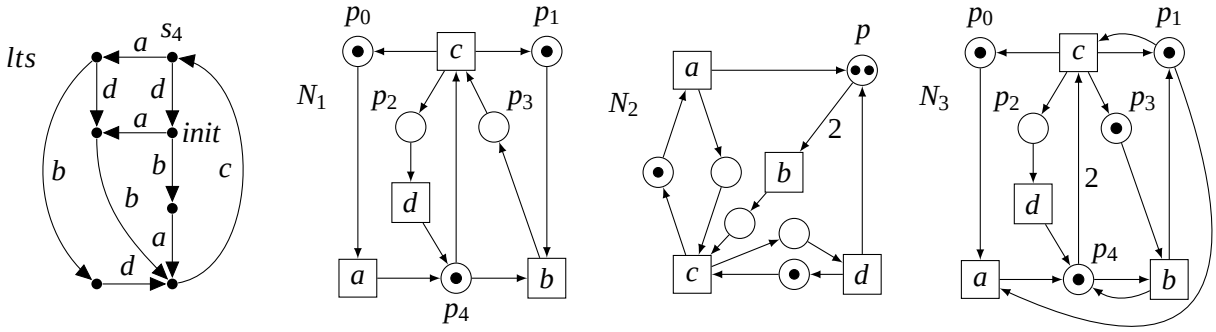

Figure 1: A persistent, reversible *lts* having the strong small cycle property with Parikh vector 1. Three Petri nets N_1, N_2, N_3 solving it are also shown. The *lts* has no marked graph solution.

allows multiset arc weights and markings to be represented readably. For switching quickly between APT and synet formats, APT contains two translation modules synet2apt and apt2synet. Third-party formats for Petri nets, such as the LoLA [16] and PNML (cf. http://www.pnml.org/) formats, are also supported.

The APT toolbox provides a large number of *modules*. If the program is started without any arguments, a full list of available modules is printed. A special module called help (already illustrated in listing 1 for the module bounded) can be used for obtaining information about a module. It can be seen that the bounded module requires a Petri net as input and optionally accepts a value k to check for k-boundedness. In listing 3 both features are exemplified. The results show that N_1 (of figure 1) is bounded,

```
.name "file name: net.apt; file content: a persistent and reversible net"
.description "A Petri net N_1 having the small cycle property"
.type LPN /* stands for Labelled Petri Net */
.places
p0 p1 p2 p3 p4 /* five places */
.transitions
a b c d /* four transitions */
.flows
a: { p0 } -> { p4 }
b: { p4, p1 } -> { p3 }
c: { p4, p3 } -> { p0, p1, p2 }
d: { 1 * p2 } -> { 1 * p4 }          // 1 * is actually redundant
.initial_marking { p0, 1 * p1, p4 } // same here
```

Listing 2: File net.apt containing N_1, as depicted in figure 1, in APT text file format.

```
$ ./apt.sh bounded net.apt
bounded: Yes
$ ./apt.sh bounded net.apt 1
bounded: No
witness_place: p4
witness_firing_sequence: [a]
```

Listing 3: Illustration of how to use the `bounded` module.
On Unix-like platforms, the shell script `apt.sh` serves as a shorthand for starting `APT`.

```
.name ""   /* file name: lts.apt (this comment was added manually) */
.type LTS  /* stands for Labelled Transition System (this comment was added manually) */
.states
s0[initial] /* [ [p0:1] [p1:1] [p2:0] [p3:0] [p4:1] ] */
s1          /* [ [p0:0] [p1:1] [p2:0] [p3:0] [p4:2] ] */
s2          /* [ [p0:1] [p1:0] [p2:0] [p3:1] [p4:0] ] */
s3          /* [ [p0:0] [p1:0] [p2:0] [p3:1] [p4:1] ] */
s4          /* [ [p0:1] [p1:1] [p2:1] [p3:0] [p4:0] ] */
s5          /* [ [p0:0] [p1:1] [p2:1] [p3:0] [p4:1] ] */
s6          /* [ [p0:0] [p1:0] [p2:1] [p3:1] [p4:0] ] */
.labels
a b c d
.arcs
s0 a s1     s0 b s2     s1 b s3     s2 a s3     s3 c s4
s4 a s5     s4 d s0     s5 b s6     s5 d s1     s6 d s3
```

Listing 4: Reachability graph of N_1, generated with `./apt.sh coverab net.apt lts.apt` and slightly edited, in order to minimise the number of lines. It is isomorphic to *lts* shown in figure 1.

but not 1-bounded. For 1-boundedness, `APT` provides a witness for the negative result, stating that after firing transition `a`, place `p4` will have more than one token on it.

The `coverability_graph` module of `APT` can be used to generate a coverability graph [7] of a Petri net. For a bounded net, this will be the reachability graph (cf. section A). Listing 4 shows the reachability graph calculated by `APT` for our running example via `./apt.sh coverability_graph net.apt`. Module names can be shortened, as long as the resulting prefix is unique. So we can also use `coverab` to call the coverability module. The initial state is always called `s0`. The correspondence between states and markings is given as a comment. The `draw` module can be used to translate the calculated graph into the DOT format used by the `GraphViz` tool (cf. `http://www.graphviz.org/`) which can then visualize the graph.

Note that, in *lts*, each small cycle contains every transition exactly once. Such a property can be examined with `APT`. The module `compute_pvs` can be used to compute the Parikh vectors of all small cycles of an lts, and the module `cycles_same_pv` checks whether all small cycles have the same Parikh vector.

3 Overview of `APT`

Four stages can be distinguished in the development of `APT`: an implementation of the necessary data structures, various analysis modules, and Petri net creator modules, described in this section, as well as, more recently, an implementation of Petri net synthesis, described in section 4 below.

Data structures of `APT`. At the heart of the `APT` toolbox sits a module system that ensures a high level of extensibility and modularity. Every module consist of an input specification, an output specification, and an algorithm. After a module has been registered with the module system, it is automatically available to be used from the command line. It is possible to create new modules by using the `Module` interface. The methods of this interface are responsible for the definition of the algorithm and the specification of parameters and return values, including their names, descriptions and types. This also includes a free text description that can include, for example, formal definitions and usage samples. Algorithms are implemented by the `run` method. Within this method, an algorithm can access the parameters that were entered by the user on the command-line. These parameters are automatically transformed into Java objects with the expected types according to the input specifications. The transformations from the textual representation to Java objects and *vice versa* happens automatically, and thus, a module can focus on working with the actual objects such as Petri nets or labelled transition systems, without needing to worry about user input / output.

For the underlying data structures implementing the objects LPN and LTS, no existing library was used, but instead, inspiration was drawn from the Petri Net API (`http://service-technology.org/pnapi/`) to design robust and versatile data structures. The main idea is the central management of data. The `PetriNet` class, respectively the `TransitionSystem` class, is used as a factory to create or delete nodes, arcs, etc. Every modification of the graph has to be done from the graph class itself, or is forwarded to it. For data storage, a compromise between memory and running time has been made. For example, the pre- and postsets of all nodes are stored by means of Java's `SoftReferences`. Hence, as long as enough memory is available, the pre- and postsets of all nodes are saved to gain a fast access to the sets. Otherwise the garbage collector of Java's Virtual Machine is allowed to delete as many pre- and postsets as necessary to achieve free memory. In this case the pre- and postsets are re-calculated and re-saved, once they are needed.

Some stand-alone analysis modules. In each case, a (negative) answer is accompanied by (counter-) examples as appropriate. The list can be extended as the need arises.

For a given finite lts (with initial state s_0),

- Check determinism, total reachability, persistence, reversibility, and the small cycle property;
- Compute weakly / strongly connected components and Parikh vectors of small cycles;
- Check (distributed) Petri net generability by two external programs, `synet` [9] and `petrify` [12].

For a given Petri net (with initial marking M_0),

- Check the existence of isolated elements, plainness, pureness, the existence of non-plain side conditions, weak / strong connectedness, coveredness by S-invariants / T-invariants, the marked graph / T-net / ON / CF / other structural properties, the BCF / BiCF properties, (k-) boundedness, (weak) liveness, persistence, reversibility, the small cycle properties as with lts, and weak / strong separability;
- Compute all connected components, the backward, forward, and incidence matrices, all side conditions, all (minimal, semipositive) S- and T-invariants, all minimal siphons / traps, the greatest common divisor of the initial marking, and **if** bounded **then** reachability graph **else** coverability graph **fi**.[1]

[1] Several of the other tasks require boundedness as a precondition, so that the boundedness check is often used as a first step.

For a given labelled Petri net with initial marking M_0 and labelling $h\colon T \to \Sigma$,

- Check whether a given word $w \in \Sigma^*$ is in the language of the net, check language equivalence, and check isomorphism and bisimulation of reachability graphs.

The tasks described in this list are obviously of very diverse degrees of complexity. One amongst them (*Given a Petri net, is it separable?*) has an unknown decidability status. Therefore, a restrictive algorithm was implemented in this module, allowing bounds to be specified for the lengths of firing sequences.

Generator modules in `APT`. These modules are useful, e.g., for benchmarking purposes (cf. section 4.2).

- Generate regular sample nets, for instance: n-bit marked graph nets, for some specification or range of n; n-philosopher nets [13]; all marked graphs with a limited number of places, transitions, and tokens.

Counterexample finding modules. These modules (understandably) suffer from runtime problems.

- For a net, check whether the preconditions of the conjecture mentioned at the end of section A are satisfied, and then check isomorphism against all marked graphs of a limited size. Do the same for a small number of randomly selected marked graphs of bigger sizes.

- Try to find intelligent extensions of an lts, such that the preconditions of the same conjecture remain satisfied. Find minimal extensions of an lts that satisfy all required properties.

4 Petri net synthesis with `APT`

The goal of net synthesis is to find an injectively labelled Petri net whose reachability graph is isomorphic to a given lts. `APT`'s `synthesize` module (a recent addition to `APT` by the second author) accepts up to three parameters. The second parameter is the transition system from which a Petri net should be synthesised and the third parameter can optionally specify where the calculated Petri net could be saved. The first parameter is a comma-separated list of properties that the produced Petri net should satisfy. Supported properties are, at present: `none`, which can be used if just a generic P/T net without special properties is needed; `pure` to synthesise a net without side-conditions; `plain` if a net without weights is required; `output-nonbranching` when a place may not have more than one transition in its post-set; `t-net` when each place may also not have more than one transition in its pre-set; `conflict-free` when each place is either output-nonbranching or its post-set is a subset of its pre-set; *k*`-bounded` if every place must never contain more than k tokens in any reachable marking; `safe` if the net should be 1-bounded; `language` if only a Petri net with the same prefix language is searched for; and `verbose` to print additional information about the calculated solution. These definitions conform to those of section A and [5, 7]. Additionally, a distributed Petri net can be requested (see below).

As an example, consider the reachability graph *lts* shown in figure 1. Let us start by just requesting any Petri net solution. This is done by running `./apt.sh synth none lts.apt`. One possible solution is shown as N_3 in the same figure. This net is similar to N_1 in the sense that both of them have reachability graph *lts*, but some structural differences can be observed. Synthesis is implemented by an algorithm [1] involving the solution of several systems of linear inequalities. These solutions give rise to a large (possibly redundant) set of regions. From these regions, `APT` selects a non-redundant but still sufficiently

```
$ ./apt.sh synthesize safe,verbose lts.apt
success: No
solvedEventStateSeparationProblems:
Region { init=1, 0:a:0, 0:b:0, 1:c:0, 0:d:1 }:
        separates event c at states [s4, s5, s6]
Region { init=0, 0:a:0, 0:b:1, 1:c:0, 0:d:0 }:
        separates event c at states [s0, s1, s4, s5]
[...]
failedStateSeparationProblems: []
failedEventStateSeparationProblems: {b=[s4]}
```

Listing 5: Failure when trying to synthesise a safe Petri net from *lts* (s_4 refers to a node in figure 1)

```
[...] .labels
      a[location="A"] b[location="B"] c[location="A"] d[location="A"]
[...]
```

Listing 6: Adding locations to the reachability graph from listing 4. Only the changes are shown.

large subset, so that the corresponding Petri net also solves *lts*, provided the latter is solvable at all. Depending on the way these inequality systems are solved, different non-redundant sets of regions may be produced. In some releases, `APT` used (and incorporated) `ojAlgo` (cf. `http://ojalgo.org/`). Later, `SMTInterpol` [11] was used. N_3 is created via `ojAlgo`; in other releases, a different solution of *lts* can and will be obtained. The implementation is exact in the sense that if any solution exists, one will be found. No further guarantees about the synthesised Petri net can be made.

As mentioned above, `APT` supports the synthesis of Petri nets with special properties. For example, suppose that we wish a synthesised net to be plain and pure. Then we can run `./apt.sh synth plain,pure lts.apt`. In this case, `APT` modifies the set of inequalities handed to a solver; the solver returns a different solution; and `APT`'s selection process constructs a set of non-redundant regions corresponding to the net N_1 shown in figure 1. The same net is calculated when `2-bounded` or just `plain` or `pure` is specified, although none of this can be guaranteed by the implementation.

If we try to synthesise a safe Petri net from *lts*, we get a failure. The corresponding arguments to `APT` and its output are shown in listing 5. This is also an example for the `verbose` option. Each calculated region of the lts corresponds to a place in the Petri net that is being synthesised. For example, the first region in the above output is { `init=1, 0:a:0, 0:b:0, 1:c:0, 0:d:1` }. This corresponds to a place with initial marking one and from which transition *c* consumes a token while *d* produces a token each time it fires. Also, this place disables the transition *c* in states s_4, s_5 and s_6, as indicated in the output shown in listing 5. Five such regions are found, but synthesising still fails, because no region can be calculated which disables event *b* in state s_4 (cf. figure 1). In the jargon, "*b* cannot be separated safely at s_4".[2]

The `synthesize` module also supports the specification of locations for transitions. If two transitions have different locations, they must have disjoint pre-sets [4]. In both Petri nets which were synthesised so far, transitions *b* and *c* always had a common place in their pre-sets. Next, we will look for a Petri net

[2]Note that `APT`'s output is nevertheless correct. Every Petri net solution must have some place *p* which prevents *b* in the marking that corresponds to s_4. Since the sequence *db* is fireable in s_4, transition *d* must produce enough tokens on *p* to enable *b*. Also *ab* is fireable, so transition *a* produces tokens on *p* as well. Finally, the firing sequence *ad* is also enabled in s_4. By the above reasoning, both *a* and *d* produce at least one token on *p*, so after *ad* that place must be marked with at least two token. Thus, no safe Petri net solution exists.

```
$ ./apt.sh word_synthesize none a,b,b,a,a,c
success: No
separationFailurePoints: a, b, [a] b, a, a, c
```

Listing 7: Example of `word_synthesize` in order to synthesise $w = abbaac$.

where b's preset is disjoint from the presets of all other transitions. Listing 6 shows how to specify this in the APT file format. If an lts contains locations, the `synthesize` module will always honour them. No special command line option to enable this is required. When synthesising a Petri net from the modified input file, the net N_2 shown in figure 1 is generated. It can be seen that the pre-sets of all transitions are disjoint in that net, even though the input file only required that transition b has no common place in its pre-set with the other transitions. In general, specifying different locations for all transitions is tantamount to requiring an ON output net.

APT also provides word synthesis. For a given word w, a Petri net with injective labelling is produced such that w and its prefixes are the only enabled firing sequences. Given a word $w = a_1a_2\dots a_n$, this module internally creates an lts $(S, \to, T, s_0)$ with $n+1$ states $S = \{s_0, s_1, \dots s_n\}$, transitions $T = \{a_1, a_2, \dots a_n\}$, and transition relation $\to = \{(s_{i-1}, a_i, s_i) \mid i \in \{1, 2, \dots, n\}\}$. Listing 7 shows an application. In the first line, APT is asked to synthesise the word *abbaac* (specified as a comma-separated list). The set of transitions is implicitly assumed to be $T = \{a, b, c\}$. No requirements are specified for the synthesised Petri net, and still, a failure occurs. The output shows that after the subword *ab*, the transition a is enabled, even though the input requires the transition b to be the only enabled transition.[3]

4.1 Some algorithmic background

By courtesy of the authors of [1], the authors were fortunate to be able to use an advance draft of [1] when implementing the `synthesize` module. Nevertheless, for the purpose of creating solutions with special properties, it was necessary to extend the theory somewhat. Some of these amendments are described (very briefly) in the following. APT contains a generic implementation that can handle all of the supported properties, and for some special cases, APT contains faster algorithms.

Formally, a region of an lts $(S, \to, T, s_0)$ is a triple $(\mathbb{R}, \mathbb{B}, \mathbb{F}) \in (S \to \mathbb{N}, T \to \mathbb{N}, T \to \mathbb{N})$ such that for all $s[t\rangle s'$ with $s \in [s_0\rangle$, $\mathbb{R}(s) \geq \mathbb{B}(t)$ and $\mathbb{R}(s') = \mathbb{R}(s) - \mathbb{B}(t) + \mathbb{F}(t)$. Essentially, $\mathbb{B}$ and $\mathbb{F}$ assign *backward and forward weights* to transitions t of an lts, so that these weights can serve as connecting arc weights between t and a place of a Petri net, and $\mathbb{R}$ assigns a token count in each marking to that place. The derived function $\mathbb{E}\colon T \to \mathbb{Z}$ defined by $\mathbb{E}(t) = \mathbb{F}(t) - \mathbb{B}(t)$ is called the *effect* of a transition t. Because the effect is zero around cycles of the lts, the functions $\mathbb{B}$ and $\mathbb{F}$ necessarily satisfy $\sum_{t \in T} \Psi(t) \cdot \mathbb{B}(t) = \sum_{t \in T} \Psi(t) \cdot \mathbb{F}(t)$ for every cyclic Parikh vector Ψ in the lts. A region is called *pure* if it satisfies $\forall t \in T\colon \mathbb{B}(t) = 0 \vee \mathbb{F}(t) = 0$.

For synthesising a Petri net from an lts, regions solving *separation problems* have to be found. There are two kinds of such problems. For each state s in which transition t is not enabled, there is an *event/state separation problem* $\mathbb{R}(s) < \mathbb{B}(t)$ that corresponds to a place preventing the transition t. For each pair of states $\{s, s'\}$ with $s \neq s'$ there is a *state separation problem* $\mathbb{R}(s) \neq \mathbb{R}(s')$ so that these states are represented by different markings. The task at hand is to find, for any given separation problem, a region

[3] This result is correct since a cannot be separated at state s_2. That is, any injectively labelled Petri net in which the word *abbaac* and all of its prefixes are fireable, must also have a firing sequence *aba*.

that solves it. A set R of regions is feasible for synthesising a Petri net if each separation problem is solved by at least one of its regions. In this case the Petri net described by R solves the lts. However, since special properties might be requested from the calculated Petri net, only regions which do not contradict these properties should be used. Some algorithms optimise the search for feasible regions but do not allow special properties to be guaranteed. Others are less efficient in general but more flexible in terms of the result. `APT` chooses an appropriate algorithm, which may depend on the result specification, as follows.

Petri net synthesis with additional properties. `APT` comes with a general algorithm supporting all properties. For this, first a region basis is calculated from the cycles of the transition system. This basis has the property that all pure regions are a linear combination of its elements. An inequality system is used for finding such a combination. For solving a specific separation problem, the initial marking $\mathbb{R}(s_0)$ and the backward and forward weights $\mathbb{B}(t)$ and $\mathbb{F}(t)$ for every transition t are variables. With these, we explicitly require for any state $s' \in S$ and enabled transition $t \in T$ that the region does not block t. This can be expressed via $s'[t\rangle \implies \mathbb{R}(s') = \mathbb{R}(s_0) + \mathbb{E}(\Psi_{s'}) \geq \mathbb{B}(t)$, where $\Psi_{s'}$ is the Parikh vector of the path from s_0 to s' in some fixed spanning tree. Then, any solution of the system describes a valid region of the lts under consideration. For separating states s and s', an additional inequality $\mathbb{R}(s) \neq \mathbb{R}(s')$ is required. Since for each place of a bounded Petri net, a complementary place can be added so that the token sum of the two places stays constant, this inequality can be softened to $\mathbb{R}(s) < \mathbb{R}(s')$. For separating transition t from state s, either $\mathbb{R}(s) - \mathbb{B}(t) < 0$ or $\mathbb{R}(s) + \mathbb{E}(t) < 0$ is used, depending on whether the resulting Petri net should be impure or pure.

Additional inequalities are added to guarantee the requested properties. When locations are specified, only transitions on the same location as t may have $\mathbb{B}(t) > 0$, i.e., may consume token from this place in the final Petri net. For all other transitions t' the equation $\mathbb{B}(t') = 0$ makes sure that no conflict between locations occurs. Calculating output-nonbranching solutions makes use of this by internally assigning a unique location to each transition. If the user asks for a plain solution, the algorithm adds $\mathbb{B}(t) \leq 1$ and $\mathbb{F}(t) \leq 1$ for every transition $t \in T$ to the inequality system. T-nets are found by requiring a plain solution where additionally the sum of all forward weights is at most one, $1 \geq \sum_{t \in T} \mathbb{F}(t)$, and the same for backward weights. If a conflict-free net should be synthesised, plainness is additionally required and the implementation first searches an output-nonbranching region and, if this fails, the corresponding inequalities are replaced with $\mathbb{E}(t) \geq 0$ for all transitions t. This ensures that the preset contains the postset of the place that corresponds to the calculated region. Finally, calculating a k-bounded Petri net requires adding an inequality $k \geq \mathbb{R}(s)$ for each state s. Because this is, so far, the only property that requires adding an inequality for each state, it is the most expensive one.

Speeding up general Petri net synthesis. If the `synthesize` module is invoked just with result specification `none`, and no locations are specified, synthesis can be made more efficient. The approach for event/state separation is to calculate a region where $\mathbb{R}(s)$ is smaller than $\mathbb{R}(s')$ for any state s' in which transition t is enabled. Then both $\mathbb{B}(t)$ and $\mathbb{F}(t)$ can be increased by the same amount (possibly introducing side conditions) so that the transition becomes separated. To find such a region, the system $\forall s' \in S\colon s'[t\rangle \implies \mathbb{E}(\Psi_s - \Psi_{s'}) < 0$ has to be solved, where the weights of the region basis are the unknowns (that is, a much smaller system has to be solved). For state separation, the regions from the region basis can be tested and used. This is because if the regions from the basis do not separate s and s', then no linear combination of the basis elements will either.[4]

Pure and pure&plain Petri net synthesis. Suppose that the result request is `pure`, or `pure,plain`

[4]Note: This algorithm and the previous one (without additional properties beside pure) are described in detail in [1].

(read conjunctively), and that again, no locations are supported. For solving state separation, if only a pure solution is requested, the previous approach can be used, because all elements of the region basis calculated there are pure regions. For separating transition t from state s, by definition, a region satisfying $\mathbb{R}(s) < \mathbb{B}(t)$ is needed. Since $\mathbb{R}$ can be calculated based on the value $\mathbb{R}(s_0)$ for the initial state and the Parikh vector Ψ_s, this is equivalent to $\mathbb{R}(s_0) + \mathbb{E}(\Psi_s) - \mathbb{B}(t) < 0$. After more simplifications, we see that we have to solve $\forall s' \in S\colon \mathbb{E}(\Psi_s - \Psi_{s'} + \mathbf{1}_t) < 0$ where $\mathbf{1}_t$ is the t-unit vector. As before, the resulting region has to be a linear combination of the region basis. If a plain Petri net should be calculated, additional constraints are added that ensure that $-1 \leq \mathbb{E}(t) \leq 1$ for all transitions t i.e., that the forward and backward weights are either one or zero.

Synthesising marked graph Petri nets. The reachability graphs of marked graphs are characterised and a special synthesis algorithm is presented in [5]. This algorithm calculates a Petri net solution directly, based on structural properties of the lts, and is implemented in `APT`. The details will not be repeated in the current paper. Suffice it to say that `APT`'s `synthesize` module automatically checks the required structural preconditions on the lts and uses the improved algorithm if it is applicable. This algorithm supports any combination of the properties `pure`, `plain`, and `t-net`, and any location specification.

Synthesis up to language equivalence. If a Petri net with the same prefix language as the input lts is needed, a so-called limited unfolding of the lts [1] is calculated. This unfolding is synthesised as usual, but without enforcing state separation.

Heuristically minimizing the number of places. A feasible set of regions could stay feasible if some regions are removed from it. This can occur because regions calculated for a specific separation problem could additionally solve other separation problems. Thus, it makes sense to remove unnecessary regions from the set of calculated regions. For this, all event/state separation and state separation problems are evaluated again in the regions found. If such a problem is solved by just a single region, that region cannot be removed from the feasible set of regions. This region is called a *required* region. Any separation problem which is solved by a required regions can be discarded. For the remaining problems which are solved by multiple non-required regions, any of these regions could be picked arbitrarily. In practice this heuristic produces Petri nets with an acceptably low number of places.[5]

4.2 Benchmarks

The performance of `APT`[6] for Petri net synthesis was compared with `synet` 2.0b [9], `petrify` 4.2 [12] and `GENET` [10] on a system running Fedora 21 with an Intel® Core™ i7-4790 CPU clocked at 3.6 GHz and with 32 GiB of memory. The `synet` tool can synthesise distributable bounded Petri nets. For `petrify`, the user can choose between some properties, for example pure, free choice and unique choice. However, `petrify` only creates safe Petri nets and employs transition splitting to ensure that a solution exists. This means that the resolution Petri nets might not be injectively labelled. With `GENET`, the result will only be bisimilar to the input. Also, this tool requires *a priori* knowledge about the maximum number of token on any place, and it resorts to transition splitting to produce solutions. Given these differences, it can be expected that `petrify` and `GENET` perform better on safe nets and worse on transition systems which have no safe solution.

Three of `APT`'s Petri nets generators were used. The `bitnet_generator` module creates a net where n

[5] This heuristic introduces nondeterminism. Alternatively, some total ordering could be imposed on regions to break ties.

[6] The latest development version was used. It can be identified by git commit id `14651f7280db255d1539` in [8].

	bit net synthesis					philosophers' net synthesis				
n	APT	APTp	synet	petrify	GENET	APT	APTp	synet	petrify	GENET
8	0.60	0.86	138.49	0.13	0.05	0.55	0.49	0.06	0.01	0.01
10	1.56	2.32	—	1.25	0.31	0.50	0.60	10.08	0.05	0.03
12	5.71	6.31	—	17.73	2.28	0.79	1.05	—	0.25	0.09
14	24.69	30.48	—	403.67	16.10	1.72	2.42	—	0.91	0.33
16	183.76	212.23	crash	—	132.13	4.49	5.21	—	4.11	1.31
18	—	—	crash	OOM	—	9.17	13.13	—	21.84	4.83
20						26.76	41.96	—	171.10	19.88
22						98.57	146.42	crash	—	123.05

Table 1: Time in seconds for synthesising a Petri net. APTp means `APT` with the `pure` parameter. Dashes indicate that the 10 minutes time limit was exceeded. For large inputs, `synet` crashed with a stack overflow and `petrify` exited with a memory allocation error.

bits can be flipped between two states, creating 2^n states in total. The `bistate_philnet_generator` model Dijkstra's philosophers problem [13] for n philosophers such that each philosopher grabs both forks in a single step and puts them back simultaneously as well. The `cycle_generator` creates a cycle consisting of n transitions and n places where k tokens are moved from one place to the next in a cyclic way. All these generators produce plain and pure nets. The first two generators and cycles with $k = 1$ are additionally safe. In this case, all contesting tools can correctly synthesise nets from the reachability graph of the generated nets, although `GENET` might produce a net which only exhibits bisimilar behaviour. For $k > 1$, transition splitting will be done by `petrify` and `GENET`.

`petrify` was used with argument `-dead`, so that it does not complain about deadlocks. `APT` was measured for general synthesis and for pure synthesis. In contrast to `petrify`, which produced similar run times in these two cases, this makes a difference for `APT`. `synet` was only benchmarked with parameter `-r`, since it performed consistently worse without this argument. `GENET` was used without any arguments. Measurements were made by generating the reachability graph of the net that the Petri net generator produced, converting the net into the input format of each tool with `APT` and then measuring the wall clock time needed by each tool to synthesise a Petri net from this graph. The time for synthesis was limited to 10 minutes via the `ulimit -t` unix command. For each input, three measurements were taken, out of which the minimal values are depicted in Tables 1 to 3.

The result for the class of bit nets are shown in the left part of table 1. It can be seen that with 18 bits, none of the tools managed to find a solution within the 10 minutes time limit. This table also shows that `APT` has a relatively high start-up cost, causing it to require more time for small inputs. Also, `APT` only slows down moderately if a pure solution is requested. Surprisingly, `synet` crashes with a stack overflow error if the input becomes too large and `petrify` runs out of memory for the reachability graph of a 17 bit (not shown) or 18 bit net. Its peak memory usage is about 1 GiB, so the system's physical memory is not exhausted. In this benchmark, `GENET` is a bit faster than `APT`.

Table 1 also contains the results for the philosophers' nets in its right part. Here `APT` outperforms `GENET`, but only for the largest inputs. Up to $n = 20$, `GENET` is consistently faster. When requesting a pure solution, `APT` becomes slower than `GENET` searching for any solution at all. When compared to to `petrify`, similar behaviour can be seen, although here the crossing point is at $n = 17$. In this experiment `APT` is still faster than `GENET` if a pure solution is requested.

n	APT	APTp	synet	petrify	GENET
100	0.44	0.45	1.12	0.02	0.28
180	1.58	1.58	8.83	0.05	1.81
260	5.44	5.45	35.99	0.10	6.34
340	16.45	16.05	102.52	0.15	17.45
420	40.55	40.90	234.59	0.23	32.99
500	83.15	83.53	475.50	0.32	62.39

Table 2: Cycle synthesis run times with cycles of size n and $k = 1$ token.

The times for the cycle nets with a single token are shown in table 2. Compared to the other examples, these nets show no concurrent behaviour and are about as large as their reachability graphs. In this benchmark, `APT` uses its implementation of the marked graph synthesis from [5]. Still, `petrify`, for reasons not known to the authors, almost needs no time at all.

size n varying, $k = 5$ tokens fixed						size $n = 5$ fixes, k tokens varying					
n	APT	APTp	synet	petrify	GENET	k	APT	APTp	synet	petrify	GENET
5	0.19	0.19	0.00	10.08	136.52	5	0.19	0.19	0.00	10.08	136.52
10	0.49	0.51	—	—	468.38	10	0.37	0.30	0.16	—	292.74
15	1.35	1.39	—	—	—	15	0.61	0.72	3.19	—	—
20	4.83	4.58	—	—	—	20	2.00	1.14	16.47	—	—
						25	2.42	2.09	93.22	—	—
						30	4.16	3.81	190.81	—	—

Table 3: Cycle synthesis run times with cycles of size n and k tokens. Left part varies size of cycle, right part varies number of token.

When synthesising cycles with $k = 5$ tokens, the cycles have to be a lot smaller. The corresponding result are shown in the left part of table 3, and it can be seen that the tools that use transition splitting need much longer. The debug output suggests that the splitting leads to an exponential increase in the state space. Also, `synet` only manages to synthesise the smallest cycle size within the time limit. In contrast to this, `APT` produces results quickly, because in this case, the marked graph synthesis algorithm performs optimally. The results for cycles of size $n = 5$ with increasing numbers of tokens are similar and can be found in the right part of the same table. The main difference is that `synet` performs a lot better when the number of tokens is increased instead of enlarging the size of the cycle.

An experiment was done by hand for cycles of size $n = 3$ with $k = 100$ tokens. In this setup, `APT` needed 0.65 seconds to find a solution, `synet` finished in 0.98 seconds and `APT` with parameter `pure` in 1.03 seconds. `GENET` ran out of memory after allocating 4 GiB in 422 seconds. After 40 minutes without any result, `petrify` was aborted. In this special case, `GENET` was also measured with parameter `-k 100`, telling it to look for 100-bounded solutions, and found one in 7.67 seconds. When the search with bounds 1 to 99 was skipped via parameters `-k 100 -min 100`, `GENET` needed only 2.20 seconds. This confirms previous intuitions that transition splitting may lead to bad run times (and, of course, to non-injectively labelled nets), but it also shows that `GENET` is sped up if *a priori* knowledge is available. Still, even for the safe case, `APT` has comparable results and has been generalised (like `synet`) to unsafe nets.

5 Concluding remarks

APT's algorithms are packaged in a single, portable archive called `apt.jar`. The idea is that a user can copy this file and run it smoothly, using his or her favourite text editors, in a local Java 7 environment, or alternatively, grab the entire `apt` directory from the repository at [8] and build a local copy of `apt.jar` using ANT. APT's performance in its other modules (for example, `coverability`) was tested against other tools (for example, `LoLA 2.0` [16]) and seems to perform worse, but not hopelessly so.[7] In general, the authors hope that all of APT's modules can be used sensibly in a classroom environment, say for a course on place/transition Petri nets and finite transition systems. They also believe that APT's more sophisticated algorithms can, in addition, be helpful to researchers in the corresponding areas.

In future, we wish to explore whether code written, say, in C++ could be incorporated into APT more tightly than just by means of exchanging text files for nets and transition systems. Also, graphical extensions will be explored cautiously (cf. [14]). However, before imposing a more powerful user interface onto APT, we would like to explore intelligent – possibly interactive – extensions. For instance, consider the algorithm testing the strong small cycle property. If no prior assumptions hold, it is nontrivial and, in general, rather time-consuming. However, suppose that the preconditions of the result mentioned at the end of section A have already been tested and are known to hold for the given lts. Then we know that the weak small cycle property also holds, and testing the strong one is much easier. (The same principle – using theory to algorithmic advantage – is behind APT's fast marked graph synthesis.) It is also planned to extend word synthesis to the prefix languages of regular languages. This is pretty straightforward, since it is well-known how to construct an lts from a regular expression. Other extensions could consist of parallelising some of the algorithms. Dennis Borde, one of the APT students, already succeeded in parallelising part of the coverability graph generation algorithm by exploiting the power of a graphics card processor running concurrently with the main processor.

Acknowledgements: The authors would like to thank the reviewers for helpful comments.

References

[1] É. Badouel, L. Bernardinello, P. Darondeau: Petri Net Synthesis. In preparation (330 pages). Springer-Verlag (2015).

[2] E. Best, P. Darondeau: A Decomposition Theorem for Finite Persistent Transition Systems. Acta Informatica 46:237–254 (2009). doi:10.1007/s00236-009-0095-6

[3] E. Best, P. Darondeau: Separability in Persistent Petri Nets. Fundamenta Informaticae 113(3-4), 179–203 (2011). doi:10.3233/FI-2011-606

[4] E. Best, P. Darondeau: Petri Net Distributability. In I. Virbitskaite, A. Voronkov (eds): PSI'11, Novosibirsk, LNCS Vol. 7162, Springer-Verlag, 1–18 (2011). doi:10.1007/978-3-642-29709-0_1

[5] E. Best, R. Devillers: A Characterisation of the State Spaces of Live and Bounded Marked Graph Petri Nets. Proceedings of LATA'14, LNCS Vol. 8370, Springer-Verlag, 161–172 (2014). doi:10.1007/978-3-319-04921-2_13

[6] E. Best, R. Devillers: Synthesis of Persistent Systems. In G. Ciardo, E. Kindler (eds): Proc. ICATPN'14, Tunis, LNCS Vol. 8489, pp. 111-129 (2014). doi:10.1007/978-3-319-07734-5_7

[7] The authors are aware of (and have tested) a multitude of other Petri net tools. Not all of them could be mentioned in this paper.

[7] E. Best, H. Wimmel: Structure Theory of Petri Nets. Proc. of the Fifth Advanced Course on Petri Nets, Rostock, 2010, K. Jensen et al. (eds): ToPNoC VII, volume 7480 of LNCS, Springer-Verlag, 162–224 (2013). doi:10.1007/978-3-642-38143-0_5

[8] D. Borde, S. Dierkes, R. Ferrari, M. Gieseking, V. Göbel, R. Grunwald, B. von der Linde, D. Lückehe, U. Schlachter, C. Schierholz, M. Schwammberger, V. Spreckels: `https://github.com/CvO-theory/apt`

[9] B. Caillaud: `https://www.irisa.fr/s4/tools/synet/`

[10] J. Carmona et al.: `http://www.cs.upc.edu/~jcarmona/genet.html`

[11] J. Christ, J. Hoenicke, A. Nutz: Proof Tree Preserving Interpolation. In N. Piterman, S. Smolka (eds): Tools and Algorithms for the Construction and Analysis of Systems, held as Part of ETAPS 2013, Springer-Verlag, LNCS Vol. 7795,124–138 (2013). doi:10.1007/978-3-642-36742-7_9

[12] J. Cortadella et al.: `http://www.cs.upc.edu/~jordicf/petrify/`

[13] E.W. Dijkstra: Hierarchical ordering of sequential processes. Acta Informatica 1(2), 115-138 (1971). doi:10.1007/BF00289519

[14] H. Saathoff: `APE` – an `APT` Editor. BSc Thesis, Universität Oldenburg (2013). The code can be found at [8].

[15] E. Teruel, J.M. Colom, M. Silva: Choice-Free Petri nets: a model for deterministic concurrent systems with bulk services and arrivals. IEEE Transactions on Systems, Man and Cybernetics, Part A, 27-1 (1997), 73-83. doi:10.1109/3468.553226

[16] K. Wolf: Distributed Verification with LoLA. Fundamenta Informaticae, 54(2-3), 253-262 (2003). Version 2.0 at `http://download.gna.org/service-tech/lola/`

A Labelled transition systems and Petri nets

An lts (labelled transition system with initial state) is a tuple $(S, \to, T, s_0)$, where S is a set of *states*; T is a set of *labels* with $S \cap T = \emptyset$; $\to \subseteq (S \times T \times S)$ is the *transition relation*; and $s_0 \in S$ is an *initial state*. A label t is *enabled* in a state s, denoted by $s[t\rangle$, if there is some state s' such that $(s,t,s') \in \to$. $s[t\rangle s'$ ($s[\tau\rangle s'$) means that s' is *reachable* from s through the execution of t (resp., of $\tau \in T^*$). By $[s\rangle$, we denote the set of states reachable from s. For $\sigma \in T^*$, the *Parikh vector* $\Psi(\sigma)$ is a T-vector where $\Psi(\sigma)(t)$ denotes the number of occurrences of t in σ. $s[\sigma\rangle s'$ is called a *cycle* if $s = s'$, and $\Psi(\sigma)$ is called *cyclic* in this case. A nontrivial cycle $s[\sigma\rangle s$ around a reachable state $s \in [s_0\rangle$ is called *small* if there is no nontrivial cycle $s'[\sigma'\rangle s'$ with $s' \in [s_0\rangle$ and $\Psi(\sigma') \lneqq \Psi(\sigma)$.

Two lts $(S_1, \to_1, T, s_{01})$ and $(S_2, \to_2, T, s_{02})$ over the same set of labels T are *language-equivalent* if their initially enabled sequences coincide, i.e., if $\forall \sigma \in T^* \colon s_{01}[\sigma\rangle \iff s_{02}[\sigma\rangle$, *isomorphic* if there is a bijection $\zeta \colon S_1 \to S_2$ with $\zeta(s_{01}) = s_{02}$ and $(s,t,s') \in \to_1 \iff (\zeta(s),t,\zeta(s')) \in \to_2$, for all $s, s' \in S_1$; and *bisimilar* if there is a relation $\beta \subseteq S_1 \times S_2$ with $(s_{01}, s_{02}) \in \beta$ and whenever $(r_1, r_2) \in \beta$ and $(r_1, t, s_1) \in \to_1$, then $\exists s_2 \in S_2 \colon (r_2, t, s_2) \in \to_2$ (and *vice versa*).

A labelled transition system $(S, \to, T, s_0)$ is called *finite* if S and T (hence also $\to$) are finite sets; *deterministic* if for any reachable state s and label a, $s[a\rangle s'$ and $s[a\rangle s''$ imply $s' = s''$; *totally reachable* if $S = [s_0\rangle$ and $\forall t \in T \exists s \in [s_0\rangle \colon s[t\rangle$; *reversible* if $\forall s \in [s_0\rangle \colon s_0 \in [s\rangle$; *persistent* if for all reachable states s and labels t, u, if $s[t\rangle$ and $s[u\rangle$ with $t \neq u$, then there is some state $r \in S$ such that both $s[tu\rangle r$ and $s[ut\rangle r$. It has the *weak small cycle property* if there is a finite set of mutually transition-disjoint Parikh vectors such that every small cycle has a Parikh vector in this set, and the *(strong) small cycle property* if every small cycle has the same Parikh vector.

A (finite, initially marked, place-transition, arc-weighted) Petri net is a tuple (P, T, F, M_0) such that P is a finite set of *places*, T is a finite set of *transitions*, with $P \cap T = \emptyset$, F is a *flow* function $F \colon ((P \times$

$T) \cup (T \times P)) \to \mathbb{N}$, M_0 is the *initial marking*, where a *marking* is a mapping $M\colon P \to \mathbb{N}$, indicating the number of *tokens* in each place. A transition $t \in T$ is *enabled by* a marking M, denoted by $M[t\rangle$, if for all places $p \in P$, $M(p) \geq F(p,t)$. If t is enabled at M, then t can *occur* (or *fire*) in M, leading to the marking M' defined by $M'(p) = M(p) - F(p,t) + F(t,p)$ (notation: $M[t\rangle M'$). The *reachability graph of* N, with initial marking M_0, is the labelled transition system with the set of vertices $[M_0\rangle$ (i.e., the states which are reachable from M_0) and set of edges $\{(M,t,M') \mid M,M' \in [M_0\rangle \wedge M[t\rangle M'\}$. If an lts TS is isomorphic to the reachability graph of a Petri net N, then we will also say that N *solves* TS. If k is a natural number and M a marking, then $k{\cdot}M$ denotes the marking with $(k{\cdot}M)(p) = k{\cdot}M(p)$ for every place p.

For a place p of a Petri net $N = (P,T,F,M_0)$, let ${}^\bullet p = \{t \in T \mid F(t,p) > 0\}$ its pre-places, and $p^\bullet = \{t \in T \mid F(p,t) > 0\}$ its post-places. N is called *(strongly/weakly) connected* if it is strongly/weakly connected as a graph; *plain* if $cod(F) \subseteq \{0,1\}$; *pure* or *side-condition free* if $p^\bullet \cap {}^\bullet p = \emptyset$ for all places $p \in P$; ON (*place-output-nonbranching*) if $|p^\bullet| \leq 1$ for all places $p \in P$; CF (*conflict-free*) if it is plain and $\forall p \in P\colon |p^\bullet| > 1 \Rightarrow p^\bullet \subseteq {}^\bullet p$; BCF (*behaviourally conflict-free*) if it is plain and for any two transitions $t,t' \in T$ with $t \neq t'$ and for every $M \in [M_0\rangle$, if $M[t\rangle$ and $M[t'\rangle$ then ${}^\bullet t \cap {}^\bullet t' = \emptyset$; BiCF (*binary-conflict-free*) if it is plain and for any two transitions $t,t' \in T$ with $t \neq t'$ and for every $M \in [M_0\rangle$, if $M[t\rangle$ and $M[t'\rangle$ then $\forall p \in P\colon M(p) \geq F(p,t)+F(p,t')$; a *marked graph* (*T-net*) if it is plain and $|p^\bullet| = 1$ and $|{}^\bullet p| = 1$ (resp., $|p^\bullet| \leq 1$ and $|{}^\bullet p| \leq 1$) for all places $p \in P$; *weakly live* if $\forall t \in T \exists M \in [M_0\rangle\colon M[t\rangle$ (i.e., there are no unfireable transitions); *k-bounded* for some fixed $k \in \mathbb{N}$, if $\forall M \in [M_0\rangle \forall p \in P\colon M(p) \leq k$ (i.e., the number of tokens on any place never exceeds k); *bounded* if $\exists k \in \mathbb{N}\colon N$ is k-bounded; *persistent* (*reversible*) if so is its reachability graph. For a number $k \in \mathbb{N}$, a net with marking $k{\cdot}M$ is called *strongly separable from* $k{\cdot}M$ if every firing sequence starting at $k{\cdot}M$ belongs to the shuffle product of k firing sequences starting at M, and *weakly separable from* $k{\cdot}M$ if the Parikh vector of every firing sequence starting at $k{\cdot}M$ is the sum of the Parikh vectors of k firing sequences starting at M.

A *labelled Petri net* has, in addition, a labelling function $h\colon T \to \Sigma$ where Σ is some set of transition labels. This induces a double labelling of the arcs of corresponding reachability graph: first, with transitions of T, and then, with labels from Σ. In case a net is labelled, the definitions of language-equivalence, isomorphism and bisimulation are the same as previously, except that they are taken with respect to Σ. If a net is unlabelled, $\Sigma = T$ is assumed implicitly (and explicitly in `APT`).

The interest of the small cycle property arises from the following result [2]: *The reachability graph of a bounded, weakly live, reversible, persistent Petri net N is finite and satisfies the weak small cycle property.* If one requires connectedness and replaces "persistent" by "ON", then the strong small cycle property can be deduced. This suggests a close relationship between persistent lts having the small cycle property and ON Petri nets, motivating a question which was raised in [4]: *If an lts is Petri net solvable, reversible, persistent, and has the small cycle property, does there always exist an ON Petri net generating it?* The answer is negative, even if the critical Parikh vector is 1 and further conditions are imposed [5]. The search for a counterexample turned out to be tedious, and was, in fact, one of the reasons for initiating `APT`. Another reason was the desire for tool support in examining further open questions, such as the following one from [3]: *Is the reachability graph of a plain, pure, bounded, reversible, persistent net with an initial marking* $K{\cdot}M$ *with* $K \geq 2$ *always isomorphic to the reachability graph of some marked graph?*