Synthesizing Petri nets from LTL specifications – An engineering approach

Dirk Fahland*

Humboldt-Universität zu Berlin, Institut für Informatik,
Unter den Linden 6, 10099 Berlin,Germany,
fahland@informatik.hu-berlin.de

Abstract. In this paper we present a pattern-based approach for synthesizing truly distributed Petri nets from a class of LTL specifications. The synthesis allows for the automatic, correct generation of humanly conceivable Petri nets, thus circumventing a manual construction of nets, or the use of Büchi automata which are not distributed and often less intuitive to understand.

1 Introduction

Coming up with an initial operational model of a distributed system that fulfils all requirements of a (temporal-logic) specification is fairly difficult and time-consuming. Even if the specification language is well-founded and suits the application domain, like the *declarative service flow language*, DecSerFlow [9], that is based on LTL, the problem persists: The available synthesis algorithms, like those used in DecSerFlow, yield a correct but hardly conceivable, and non-distributed model [1].

A faithful modeling of workflows as mathematical abstractions of actual work procedures requires a notion of local tasks, local resources, and their causal and temporal dependencies. Such notions naturally occur in models of distributed computations like process algebras or Petri nets, which are successfully applied in this domain [3, 10]. A crucial aspect of a workflow model is that it can be understood by a person looking at, typically, its graph representation. This requirement enables experts in the domain of the workflow – like business processes or disaster management – to build and refine these models. This paper deals with the challenge of synthesizing such an understandable model from a temporal-logic specification.

We present a structurally compositional approach to synthesize Petri nets from a chosen class of LTL formulae. This class is generated from a set of predefined LTL constraints and closed under conjunction; as an example, we will use the constraints provided by DecSerFlow. Following the 'divide et impera' principle, we propose a behaviorally equivalent Petri net pattern for each such LTL constraint. A conjunction of constraints translates to a merge of the corresponding Petri net patterns.

* The author's work is funded by the DFG-Graduiertenkolleg 1324 "METRIK".

Due to the local nature of each pattern, the synthesized Petri net provides truly concurrent transitions and places wherever the LTL specification makes no statement about their ordering. The constraint-implementing Petri net patterns are very small and each net element has a meaningful interpretation in terms of the application domain. Therefore, the resulting Petri net is humanly understandable and can apprehensibly be refined and analyzed. We will explain our approach in more detail in Sect. 2. We conclude in Sect. 3 together with giving some hints on analysis, and discussing ways to generalize our approach.

2 Pattern-based synthesis of Petri nets from LTL specifications

We will use this section to explain the underlying principles of our synthesis algorithm for Petri nets from LTL specifications (or more precisely DecSerFlow specifications), and provide arguments for its correctness. Beforehand, we give a brief and rather informal introduction the domain of our synthesis, a class of LTL specifications, and we recall the codomain, an extended class of Petri nets.

LTL specifications. LTL is a modal logic that allows to specify the future of paths in terms of a set *Prop* of atomic propositions. The set $\mathcal{F}_{\mathrm{LTL}}(\mathit{Prop})$ of LTL formulas is the least set containing *Prop* and that is closed under boolean connectives, the unary temporal operators $\bigcirc$ (next-state), $\Box$ (always), $\Diamond$ (eventually) and the binary operator $\mathcal{U}$ (until). [7, 5]

We confine ourselves to a special subclass $\mathcal{F}_{\mathrm{WF}}(\mathit{Tasks})$ of LTL: Firstly, let *Prop* contain only propositions $\mathit{task} = A$ where A is the name of a task in a workflow; let *Tasks* be a finite and fixed set of such names. The proposition $\mathit{task} = A$ holds in a state if A is the next task to be executed. Secondly, let $\mathcal{F}_{\mathrm{WF}}$ be a finite set of parameterized LTL formulae, called *constraint templates* in the following. Each template $\psi \in \mathcal{F}_{\mathrm{WF}}$ comes with a vector $\mathbf{x}$ of parameters that can be instantiated, $\psi[\mathbf{x} \mapsto \mathbf{v}]$, to values $\mathbf{v}$ like task names or numbers. Our class $\mathcal{F}_{\mathrm{WF}}(\mathit{Tasks})$ of formulae under consideration is generated by instantiating constraint templates and is closed under conjunction. We evaluate a formula $\phi = \bigwedge_{i=1}^{n} \psi_i[\mathbf{x}_i \mapsto \mathbf{v}_i] \in \mathcal{F}_{\mathrm{WF}}(\mathit{Tasks})$ only on *feasible* paths in which in each state holds exactly one proposition. The *behavior* that is specified by ϕ is the set of all feasible paths that satisfy ϕ.

Petri nets. We found that a faithful synthesis of such formulae benefits from *inhibitor reset Petri nets*, IR nets for short [2]. (We assume the reader to be familiar with Petri nets. A detailed introduction can, for instance, be found in [8].) An IR net $N = (P, T, F, I, R, m^0)$ is a place/transition net that contains two additional kinds of arcs: If a place $p \in P$ is connected to a transition $t \in T$ by a *reset arc*, firing t removes all tokens from p while p does not restrict the firing of t. An *inhibitor arc* from p to t disables t if p is marked, firing t has no effect on p. In the following, the *behavior* of a Petri net is the set of firing sequences it generates.

Principles of the synthesis. The result of our synthesis operationalizes each task $A \in \mathit{Tasks}$ as a transition task_A. Their firing will be constrained according to the specification using a *parameterized Petri net pattern* $N(\psi)$ for each likewise parameterized constraint template $\psi \in \mathcal{F}_{\mathrm{WF}}$. For the moment assume such an behaviorally equivalent $N(\psi)$ for each template ψ. Instantiating a constraint, $\phi = \psi[\mathbf{x} \mapsto \mathbf{v}]$, then translates to instantiating the corresponding net, $N(\phi) = N(\psi)[\mathbf{x} \mapsto \mathbf{v}]$. We operationalize the conjunction of two formulae, $\phi_1 \wedge \phi_2$, by merging the nets, $N(\phi_1) \bowtie N(\phi_2)$, where the *merge* operator $\bowtie$ is formally the union of places, transitions, arcs and initial markings. The synthesis algorithm therefore computes for a given specification $\phi = \bigwedge_{i=1}^{n} \psi_i[\mathbf{x}_i \mapsto \mathbf{v}_i] \in \mathcal{F}_{\mathrm{WF}}(\mathit{Tasks})$ the net $N(\psi_1)[\mathbf{x}_1 \mapsto \mathbf{v}_1] \bowtie N(\psi_2)[\mathbf{x}_2 \mapsto \mathbf{v}_2] \cdots \bowtie N(\psi_n)[\mathbf{x}_n \mapsto \mathbf{v}_n]$.

In the remainder of this section, we will cover the three remaining issues of our approach: (1) a proper correspondence between the behavior given by an LTL formula and the behavior of a Petri net, (2) the implementation of the empty specification, and (3) the sound definition of $N(\psi)$ for each constraint template $\psi \in \mathcal{F}_{\mathrm{WF}}$.

Finite and infinite behavior, and liveness properties. The correspondence between the behavior of Petri nets and LTL formulae is straight forward: A state in which $\mathit{task} = A$ holds corresponds to the firing of transition task_A. The correspondence of a path and a firing sequence then follows.

In principle, LTL formulae are evaluated over infinite paths. However, LTL formulae can be divided into two classes: Firstly, the class of formulae that, if evaluated to *true* on an infinite path π, also evaluates to *true* on a finite prefix of π; this class includes safety properties and finite liveness properties. And secondly the class of formulae that are satisfied on infinite paths only; this class includes infinite liveness and infinite fairness properties that, for instance, demand infinitely many occurrences of a task. For the scope of this paper we restrict $\mathcal{F}_{\mathrm{WF}}$ to the first class. Then we may define the behavior of a formula $\phi \in \mathcal{F}_{\mathrm{WF}}(\mathit{Tasks})$ to be the set of all finite, feasible paths satisfying ϕ.

This allows us to define an *acceptance criterion* in the synthesized Petri net that accepts a finite firing sequence iff the corresponding path satisfies the original LTL formula. Therefore, we add the Petri net pattern N_{end} containing an empty place final together with its only pre-transition end, see Fig. 1(a). A firing sequence will be accepting iff the marking that is reached by this firing sequence has a token on final. The Petri net patterns $N(\psi)$ for $\psi \in \mathcal{F}_{\mathrm{WF}}$ will restrict the firing of end; for instance, a liveness property $\Diamond\, \phi$ will result in a pre-place of end that becomes marked once ϕ has been satisfied.

Implementing the empty specification. The empty LTL specification allows arbitrary behavior; more precisely, the empty specification is satisfied by any path evaluating atomic propositions from *Prop*. This means in our case that the empty specification is satisfied by arbitrary occurrences of tasks from the set *Tasks*. The Petri net that exhibits the same behavior is the net that contains,

for each $A \in \mathit{Tasks}$, a transition task_A together with a marked place pre_A being pre- and post-place of task_A. Let $N(\mathit{Tasks})$ denote this net, see Fig. 1(b).

The net $N(\mathit{Tasks}) \bowtie N_{\mathsf{end}}$ will be the 'canvas' for our synthesis algorithm. It provides the same behavior as the empty specification according to our acceptance criterion. Conjoining an LTL constraint ϕ translates in the Petri net to merging with $N(\phi)$ that restricts the firing of the transitions of our canvas according to ϕ. Hence, given a specification $\phi = \bigwedge_{i=1}^{n} \psi_i[\mathbf{x}_i \mapsto \mathbf{v}_i] \in \mathcal{F}_{\mathrm{WF}}(\mathit{Tasks})$, our synthesis computes the Petri net $N(\phi) =_{\mathrm{df}} N(\mathit{Tasks}) \bowtie N_{\mathsf{end}} \bowtie N(\psi_1)[\mathbf{x}_1 \mapsto \mathbf{v}_1] \bowtie N(\psi_2)[\mathbf{x}_2 \mapsto \mathbf{v}_2] \cdots \bowtie N(\psi_n)[\mathbf{x}_n \mapsto \mathbf{v}_n]$ in linear time.

Deriving Petri net patterns. The remaining issue in our synthesis algorithm is the definition of an implementing Petri net pattern $N(\psi)$ for each constraint template $\psi \in \mathcal{F}_{\mathrm{WF}}$. The patterns depend on the chosen class of template formulae and each needs to be derived manually – this is the engineering part in our approach. We will explain the definition of three patterns from DecSerFlow [9] and provide arguments for their correctness.

A safety property. We start with the constraint $\mathsf{precedence}(A, B) =_{\mathrm{df}} (\Diamond\ \mathit{task} = B) \rightarrow [\neg(\mathit{task} = B)\ \mathcal{U}\ \mathit{task} = A]$ which formalizes that if B is ever going to occur, it must be preceded by an A. An implementing Petri net pattern for this safety property must allow any non-violating path and prevent any violating path.

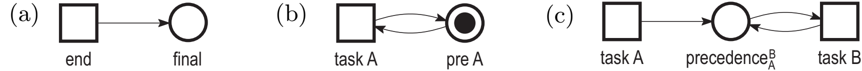

Fig. 1. The nets (a) N_{end}, (b) the building block of $N(\mathit{Tasks})$, and (c) $N(\mathsf{precendence})(A, B)$.

Our chosen solution, pattern $N(\mathsf{precedence})(A, B)$ in Fig. 1(c), relates the transitions task_A and task_B: We add an empty pre- and post-place $\mathsf{precedence}_A^B$ to task_B, to disable B initially. By making $\mathsf{precedence}_A^B$ a post-place of task_A, the occurrence of A enables B. From the logic's perspective, $\mathsf{precedence}_A^B$ gets marked iff the right-hand side of the implication has been satisfied. This gives rise to the *frame condition* of $N(\mathsf{precedence})(A, B)$ that no other pattern may change the tokens on $\mathsf{precedence}_A^B$. Please note that $\mathsf{precedence}_A^B$ is unbounded; we will discuss this aspect at the end of this section.

A liveness property. Our example of a liveness property is $\mathsf{response}(A, B) =_{\mathrm{df}} \Box\ (\mathit{task} = A \rightarrow (\Diamond\ \mathit{task} = B))$ which requires that any occurrence of A is followed by an occurrence of B. An implementing Petri net cannot prevent violating behavior of this liveness property (an occurrence of A) but must allow all behavior and provide a check for satisfaction (a subsequent occurrence of B).

The pattern $N(\mathsf{response})(A, B)$ in Fig. 2(a) does the job: We add an empty post-place $\mathsf{response}_A^B$ to task_A and add a reset arc (dashed arc) from $\mathsf{response}_A^B$

to task_B. Firing task_A will add a token to that place, firing task_B will remove all tokens. Logically, the place $\mathsf{response}_A^B$ is marked iff the implication evaluates to *false*. An inhibitor arc (arc with a black dot) from $\mathsf{response}_A^B$ to end allows end to fire only if the implication (and hence the entire constraint) is satisfied; this meets our acceptance criterion defined above. Again, the pattern's frame condition requires that no other pattern changes tokens on $\mathsf{response}_A^B$.

Counting and induction. The patterns $\mathsf{ex}_i(A) =_{\mathrm{df}} \Diamond\ (task = A \wedge \bigcirc\ \mathsf{ex}_{i-1}(A))$ and $\mathsf{ex}_0(A) = true$ request that task A occurs at least i times. An implementing pattern needs to represent the levels of recursion. Since it is a liveness property, it has to extend the pre-set of end.

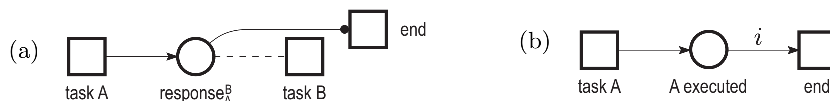

Fig. 2. The nets (a) $N(\mathsf{response})(A, B)$, and (b) $N(\mathsf{ex})_i(A)$.

We have chosen to add an empty post-place $\mathsf{executed}_A$ to task_A together with an arc of weight i from that place to end as shown in $N(\mathsf{ex})_i(A)$ in Fig. 2(b). We assume the frame condition that no other transition modifies the number of tokens on $\mathsf{executed}_A$. Then the number of tokens on that place is the number of occurrence of A and is also the number of recursion steps taken in the formula. Hence i firings of task_A yield i tokens which satisfies the constraint and enables end.

IR nets and unboundedness. The patterns that we explained above introduce unbounded places and use inhibitor and reset arcs. This is a consequence of our minimalistic approach to use exactly one transition per task. While it prohibits a direct application of standard analysis techniques, the synthesized Petri net is a structurally minimal blue-print for the implementation. By removing arcs like the loop in $N(\mathsf{precedence})(A, B)$ of Fig. 1(c), or by adding further net elements, the system can apprehensibly be refined. In [4], we defined a bisimilar place/transition net pattern for each IR net pattern; the synthesized Petri net can then be analyzed using standard techniques.

3 Conclusion and outlook

We sketched a linear-time synthesis algorithm for Petri nets from a well-defined subclass of LTL formulae. The synthesis merges small Petri nets, each implementing an LTL constraint that is a conjunct in a system specification. The synthesized Petri net is equivalent to the specification in the sense that each accepting firing sequence that marks the dedicated place final satisfies the specification and vice versa. The result of the synthesis is the starting point for refinement, analysis, and a correct implementation of the specification in terms

of a distributed system. We are not aware of another result that solves this problem. The work that comes closest to ours restricts a Petri net's behavior through merging with annotated Petri net patterns that implement constraints [6].

In [4], we demonstrate in a case study the automatic synthesis of a Petri net workflow model (11 transitions, 29 places) from a DecSerFlow specification and its analysis using model-checking. We also demonstrate how the analysis results can be related back to the specification level for refining the specification. This is made possible by our compositional approach: Each net element of the synthesized net was introduced by some Petri net pattern due to a specific constraint.

So far, we explained an engineering approach to synthesis as the modeler needs to provide a correct Petri net pattern for each LTL constraint that occurs in the specification. We have shown some patterns for constraints from the specification language DecSerFlow. The arguments for the correctness of the patterns suggest that our approach might carry further than on pre-chosen constraints and patterns only. There is a tight relation between transitions, places and markings on one side, and propositions and logical operators on the other side. We are currently searching for further correspondences between the logical connectives, and operations on Petri nets. Though, we expect that the synthesis might work only on a subclass of formulae, or may need further Petri net constructs, or that LTL is not the appropriate domain for this synthesis in general.

References

1. C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.
2. C. Dufourd, A. Finkel, and Ph. Schnoebelen. Reset nets between decidability and undecidability. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP'98*, volume 1443 of *LNCS*, pages 103–115, 1998.
3. S. Dustdar, J.L. Fiadeiro, and A.P. Sheth, editors. *Proceedings of BPM 2006*, volume 4102 of *LNCS*. Springer, 2006.
4. D. Fahland. Towards analyzing declarative workflows. In J. Koehler, M. Pistore, A.P. Sheth, P. Traverso, and M. Wirsing, editors, *Autonomous and Adaptive Web Services*, number 07061 in Dagstuhl Seminar Proceedings. IBFI, Schloss Dagstuhl, Germany, 2007.
5. D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, New York, NY, USA, 1980. ACM Press.
6. N. Lohmann, P. Massuthe, and K. Wolf. Behavioral constraints for services. In *Proceedings of BPM 2007*, LNCS. Springer-Verlag, September 2007. accepted.
7. A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.
8. W. Reisig. *A Primer in Petri Net Design*. Springer Compass International, 1992.
9. W.M.P. van der Aalst and M. Pesic. DecSerFlow: Towards a truly declarative service flow language. In *WS-FM*, pages 1–23, 2006.
10. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.