A New Product Construction for the Diagnosability of Patterns in Time Petri Net
Éric Lubat, Silvano Dal Zilio, Didier Le Botlan, Yannick Pencolé, Audine Subias

To cite this version:
Éric Lubat, Silvano Dal Zilio, Didier Le Botlan, Yannick Pencolé, Audine Subias. A New Product Construction for the Diagnosability of Patterns in Time Petri Net. 59th Conference on Decision and Control (CDC) 2020, Dec 2020, Jeju Island (virtual conference), South Korea. 10.1109/CDC42340.2020.9303826. hal-02989834

HAL Id: hal-02989834
https://hal.laas.fr/hal-02989834
Submitted on 5 Nov 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
A New Product Construction for the Diagnosability of Patterns in Time Petri Net

Éric Lubat$^1$ and Silvano Dal Zilio$^1$ and Didier Le Botlan$^1$ and Yannick Pencolé$^1$ and Audine Subias$^1$

Abstract—We propose a method to decide the diagnosability of patterns in labeled Time Petri nets (TPN) that gracefully extends a classic approach for the diagnosability of single faults. Our approach is based on a new technique for computing the language intersection of TPN and on an associated extension of the State Class Graph construction. Our approach has been implemented and we report on some experimental results.

I. INTRODUCTION

Diagnosability is a basic property of Discrete Event Systems that relates to the “observability” of concealed events. Basically, it means that every failure (a distinct instance of unobservable event) can be eventually detected after a finite number of observations. In this work, we are interested in the diagnosability of systems modeled using labeled Time Petri nets (TPN); an extension of Petri nets in which we can associate timing constraints to transitions [1]. This means that we take into account the date at which events are observed and that we want to detect failures in a bounded time. We are also interested in detecting patterns [2] in the occurrence of unobservable events. Our motivation here is to handle a broad class of diagnosis problems in a unified way, such as dealing with permanent or intermittent faults; fault sequences; etc.

We tackle the diagnosability problem using model-checking techniques; which means that we reduce diagnosability to the problem of checking the validity of a temporal logic formula on a finite-state model. This approach relies on a new class of TPN, called Product Time Petri Net (PTPN), that extends a model introduced in [3]. The idea is to enrich Petri nets with a composition operator ($\times$), semantically equivalent to synchronous product, that can be used between transitions with incompatible timing constraints. Basically, given two TPN $N$ and $N'$, the product $N \times N'$ is a net where every “observable event” from $N$ fires synchronously with a similar event from $N'$; meaning together and at the same date. We show in [3] that it is possible to extend the classical State Class Graph construction to this model. This gives an efficient method for building a finite-state representation of the traces in the intersection of (the timed language of) two or more TPN.

This paper makes several contributions. First, we describe how to use PTPN to decide the diagnosability of a system. Our approach extends the classical techniques based on the twin-plant construction and relies on an LTL model-checker. We also describe a direct, on-the-fly algorithm for diagnosability. Next, we show that we can define a notion of pattern diagnosability as a straightforward extension of the single fault problem. Our approach has been implemented in a tool called Twina [3].

The rest of the paper is organized as follows. In Section II, we discuss some previous works on diagnosability, with a focus on methods that consider Petri nets and their extensions. Section III gives technical details on TPN, Product TPN and their semantics. We describe our method for checking diagnosability in Sect. IV and briefly report on some experimental results before concluding.

II. RELATED WORK

The problem of fault diagnosis was introduced by Sampath et al [4] and is well studied in the context of discrete event systems [5], [6]. The problem has also been studied on timed models, see for instance the work of Tripakis [7] with Time Automata. In these works, diagnosability is often reduced to properties on the trace language of systems and their composition. Several works address the problem of fault diagnosis for labeled Petri nets (PN), without time [8]. There are some reasons to justify using PN instead of finite state automata in this context. For example, PN are well-suited for modeling notions such as concurrency or causality in a very compact way; they provide several notions of composition that help compositional reasoning; there is a large choice of tools for their analysis and verification; etc.

A. Diagnosability of Petri Nets

Diagnosability is a decision problem related to fault diagnosis. A system is diagnosable if we can always detect that a fault has occurred using only the record of the observable events (and in a finite number of steps) [8]. There are several efficient methods for checking the diagnosability of single faults in PN [9]. We can roughly divide existing techniques in two groups: (1) those trying to find a critical pair; and (2) “diagnoser-based” methods, trying to find executions with uncertain states. We explain these terms below.

The idea with critical pair is to execute two copies of the system, in lock step on their observable labels. A critical pair is an infinite trace where one copy has a fault and not the other; and a system is diagnosable if it has no critical pairs. The twin-plant method of [10] is representative of this group. The drawback of this approach is that we may have more states in the twin-plant than in the system. An advantage is that this method is conceptually simple.

Methods that use uncertain states build a state graph (an automaton) that is deterministic and only contains active
states and observable events. Each state is tagged with a diagnosability information: Normal, Certain, or Uncertain; where Uncertain means that we can reach the state both through a fault or without. In this case, a system is diagnosable if every state eventually leads to a Normal or Certain one, see e.g. [11]. A drawback is that we build a deterministic automaton that may be much bigger than the original (non-deterministic) state graph. Something we observe in practice.

B. Diagnosability of Time Petri Nets

Unfortunately, many of the techniques proposed for PN are not suited when we consider time constraints. One problem is that the state space of a TPN is typically infinite when we use a dense time model; that is when time delays can be arbitrarily small. Therefore we need to work on an abstraction of the transition system. A solution to this problem was proposed in [12], where the authors define a state space abstraction based on State Class Graph (SCG). Basically, a state class captures a convex set of constraints on the time at which transitions can fire. This approach is used in several model-checking tools, such as Tina [13].

Another problem with TPN is that we cannot build the composition of two transitions that have non-trivial (meaning different from $[0, \infty)$) time constraints. Therefore, it is not possible to syntactically define a “twin-plant” from a TPN. To solve this problem, we propose to use Product TPN (PTPN), a new model where we can define “groups of transitions” that should fire synchronously.

Some works address the diagnosability of TPN [14], [15], [16], but all rely on some kind of post-processing phase. While they propose substantially different methods, they all rely on a variation of the SCG construction of [12]. The approach in [16] starts by building a Modified SCG that over-approximate the possible (timed) executions. The system is diagnosable if no critical pair is found at his point. Indeed, time can only limit executions, not add new behavior. If a candidate critical pair is found, it is necessary to solve a Linear Programming problems (LPP) to check whether this scenario is feasible. This approach has several limitations. In particular, it may require to solve a large number of LPP. The other approaches are diagnoser-based and suffers from the drawbacks mentioned earlier. In [14], the authors define a notion of Augmented State Class (ASC) graphs, which are SCG with diagnosability information, and use a method to split time intervals in order to only keep deterministic paths in the ASC graph. The interval splitting phase may create a large number of new active states that can lead to a state explosion problem. The approach in [15] relies on a combination of SCG and an enumeration of all the firing sequences between active states.

We propose a direct extension of the twin-plant construction to the case of TPN, without any post-processing of traces. We use this method to decide the single fault diagnosability problem on TPN and show that it can be simply extended to decide the diagnosability of patterns. An advantage of our approach is that we can adapt it easily to several extensions of TPN: priorities between transitions; inhibitor arcs; capacity arcs; etc.

III. Time Petri Nets, Product TPN and other Technical Background

A Time Petri Net (TPN) is a net where each transition, $t$, is associated with a (static) time interval $I(t)$ that constrains the time at which it can fire. A transition is enabled when there are enough tokens in its input places. Once enabled, transition $t$ can fire if it stays enabled for a duration $\theta$ that is in the interval $I(t)$. In this case, $t$ is said time enabled.

A TPN is a tuple $\langle P,T,\text{Pre, Post}, m_0, I_s \rangle$ in which: $\langle P,T,\text{Pre, Post} \rangle$ is a net (with $P$ and $T$ the set of places and transitions); $\text{Pre, Post} : T \rightarrow P \rightarrow \mathbb{N}$ are the preconditions and postconditions functions; $m_0 : P \rightarrow \mathbb{N}$ is the initial marking; and $I_s : T \rightarrow I$ is the static interval function. We use $\mathbb{I}$ for the set of all possible time intervals. To simplify our presentation, we only consider the case of closed intervals of the form $[l,h]$ or $[l,\infty[$, but our results can be extended to the general case.

We consider that transitions can be tagged using a countable set of labels, $\Sigma = \{a,b,\ldots\}$. We also distinguish the special constant $\varepsilon$ (not in $\Sigma$) for internal, silent transitions. In the following, we use a global labeling function $L$ that associates a unique label in $\Sigma \cup \{\varepsilon\}$ to every transition. The alphabet of a net is the collection of labels (in $\Sigma$) associated to its transitions. To reason about diagnosability, we also need to partition $\Sigma$ into two sets: one for observable, the other for unobservable events. We will often use $L$ to refer to the set of observable events.

We propose an extension of TPN in which it is possible to fire several transitions “synchronously”. A Product TPN is the composition $(N,R)$ of a net $N$, with transitions $T$, and a (product) relation, $R$, that is a collection of firing sets $r_1, \ldots, r_n$ included in $T$ (hence $R \subseteq \mathcal{P}(T)$, the powerset of $T$). The idea is that all the transitions in an element $r$ of $R$ should be fired at the exact same time. As a consequence, two transitions in $r$ should have the same labels (we should use $L(r) = a$ to say they have a common label $a$) and not interfere with each other (they should not share a common input place).

Definition 1 (PTPN). A Product TPN $(N,R)$ is the pair of a net $N = \langle P,T,\text{Pre, Post}, m_0, I_s \rangle$ and a product relation $R \subseteq \mathcal{P}(T)$ such that, for every firing set $r$ in $R$, transitions in $r$ are independent and compatible: if both $t_1$ and $t_2$ are in $r$ then $L(t_1) = L(t_2)$ and for every place $p$ in $P$, $\text{Pre}(t_1)(p) > 0 \Rightarrow \text{Pre}(t_2)(p) = 0$.

Next, we define the behavior of nets. Apart from the effect of the firing sets, the following definitions are quite standard, see for instance [17], [18].

A. A Semantics Based on Firing Domains

A marking $m$ of a net $\langle P,T,\text{Pre, Post} \rangle$ is a mapping $m : P \rightarrow \mathbb{N}$ from places to natural numbers. A transition $t$ in $T$ is enabled at $m$ if and only if $m \not\succ \text{Pre}(t)$, where $\not\succ$ is the pointwise comparison between functions.
A state of a PTPN is a pair \( s = (m, \varphi) \) in which \( m \) is a marking, and \( \varphi : T \rightarrow \mathbb{I} \) is a mapping from transitions to time intervals, also called firing domains. Intuitively, if \( t \) is enabled at \( m \), then \( \varphi(t) \) contains the dates at which \( t \) can possibly fire in the future. For instance, when \( t \) is newly enabled, it is associated to its static time interval \( \varphi(t) = I_s(t) \). Likewise, a transition \( t \) can fire immediately only when \( 0 \) is in \( \varphi(t) \) and it cannot remain enabled for more than its timespan, i.e. the maximal value in \( \varphi(t) \).

For a given delay \( \theta \in \mathbb{Q}_{\geq 0} \) and \( \epsilon = [l, h] \) in \( \mathbb{I} \), we denote \( l - \theta \) the time interval \( \epsilon \) shifted by \( \theta \). This operation is defined only when \( \theta \leq h \), in which case \( l - \theta = \max(0, l - \theta) \).

By extension, we use \( \varphi - \theta \) for the partial function that associates the transition \( t \) to the value \( \varphi(t) - \theta \). This operation is useful to model the effect of elapsing time on a state.

The semantics of a PTPN is a (labeled) Kripke structure, or Time Transition System (TTS), \(<(S, S_0, \rightarrow)>\), with two possible kinds of actions: either \( s \rightarrow s' \) meaning that a set of transitions \( t \) with label \( \alpha \) is fired from \( s \); or \( s \rightarrow_{\theta} s' \), with \( \theta \in \mathbb{Q}_{\geq 0} \), meaning that we let a duration \( \theta \) elapse from \( s \). A transition \( t \) can fire from state \( (m, \varphi) \) if \( t \) is enabled at \( m \) and finable instantly. When we fire a set of transitions \( \{t_1, \ldots, t_n\} \) from state \( (m, \varphi) \), a transition \( k \) (with \( k \neq t \)) is said to be persistent if \( k \) is also enabled in the marking \( m - \sum_{t \in T} \text{Pre}(t) \), that is if \( m - \sum_{t \in T} \text{Pre}(t) \geq \text{Pre}(k) \).

The other transitions enabled after firing \( r \) are called newly enabled.

Definition 2 (Semantics of PTPN). The semantics of a PTPN \((N, R), \text{ with } N = <P, T, \text{Pre}, \text{Post}, m_0, I_s>\), is the TTS \(<(S, S_0, \rightarrow)>\), also denoted \([N, R]\), where \( S \) is the smallest set containing \( s_0 \) and closed by \( \rightarrow \) such that:

- the initial state is \( s_0 = (m_0, \varphi_0) \) where \( \varphi_0 \) is the firing domain such that \( \varphi_0(t) = I_s(t) \) for every \( t \) enabled at \( m_0 \);
- the state relation \( \rightarrow \subseteq S \times (\Sigma \cup \{\epsilon\} \cup \mathbb{Q}_{\geq 0}) \times S \) is such that for all state \((m, \varphi) \in S\):

  (i) if \( r \in R \) with labels \( a \) and \( t \) is enabled at \( m \) and \( 0 \in \varphi(t) \) for all \( t \in r \), then \((m, \varphi) \xrightarrow{a} (m', \varphi') \) where \( m' = m - \sum_{t \in r} \text{Pre}(t) + \sum_{t \in r} \text{Post}(t) \) and \( \varphi' \) is a firing function such that \( \varphi'(k) = \varphi(k) \) for any persistent transition and \( \varphi'(f) = I_s(f) \) elsewhere.

  (ii) if \( \varphi(t) - \theta \) is defined for all \( t \) enabled at \( m \) then \((m, \varphi) \xrightarrow{\theta} (m, \varphi - \theta) \).

Transitions in the case (i) above are called discrete; those labelled with delays (case (ii)) are the continuous, or time elapsing, transitions.

A Product TPN \((N, R)\) allows to fire multiple transitions simultaneously, constrained by the relation \( R \). Therefore TPN form a natural subset of PTPN, the one where every firing set has only one transition. More precisely, we can always interpret a TPN \( N \) with transitions \( \{t_1, \ldots, t_n\} \) as the PTPN \((N, R_N)\), where \( R_N \) is the collection of singleton \( \{(t_1), \ldots, (t_n)\} \). In the following, we often omit the product relation in a PTPN when it is not needed, or obvious from the context. We should also simply use the term net, or the symbol \( N \), to refer to a Product TPN.

B. Executions, Observations and Traces

An execution of a net \( N \) is a sequence of actions in its semantics, \([N]\), that starts from the initial state. It is a time-event word \( \sigma_1 \ldots \sigma_n \) over the alphabet containing both labels \((a, b, \ldots) \in \Sigma\) and delays \((\theta \in \mathbb{Q}_{\geq 0})\), where we do not record silent transitions. In the following, we simplify executions in order to avoid the occurrence of two successive delays; this can be achieved by defining a structural congruence relation between sequences, \( \equiv \), such that \( \theta' \equiv (\theta + \theta') \). This means that we can always consider executions of the form \( \sigma_0 a_0 \theta_1 a_1 \ldots \), that alternate between delays and labels. In this case, we say that the date of the event \( a_k \) in \( \sigma \) is \( \theta_0 + \ldots + \theta_k \).

Given a set of observable labels \( L \subseteq \Sigma \), the L-observation for an execution \( \sigma = \alpha_1 \ldots \alpha_k \) is the sequence \( \text{obs}_L(\alpha_1) \ldots \text{obs}_L(\alpha_k) \) such that \( \text{obs}_L(\alpha) = \alpha \) when \( \alpha \in \mathbb{Q}_{\geq 0} \cup L \) and \( \text{obs}_L(\alpha) = 0 \) otherwise. Hence \( \text{obs}_L(\sigma) \) is an execution that contains only the observable events in \( \sigma \), in the same order and at the same date than in \( \sigma \).

By contrast, a trace is the untimed word obtained from an execution when we keep only the discrete actions. Then the language of a TPN is the set of all its (finite) traces. By definition, the language of a TPN is prefix-closed; and it is regular when the net is bounded. The State Class Graph construction of [12] provides an effective method for computing a finite representation of the traces in a bounded TPN. We can do the same with Product TPN using the SCG construction defined in a previous work [3].

We can illustrate our definitions with a simple example (see Fig. 1). Executions for the net \( N_1 \) (left) are sequences of the form \( \sigma_0 a_0 b \) (and their prefix), provided that \( \theta_1 \geq 1 \); the \( \{b\}\)-observations are of the form \( b, \) with \( \theta \geq 1 \); and the only maximal trace in \( N_1 \) is \( a b \).

We can define the set of observations that are common to two nets, \( N_1 \) and \( N_2 \), using a classic product operation between transition systems. Assume \( K_1 = <S_1, s_1', \alpha_1_1> \) and \( K_2 = <S_2, s_2', \alpha_2_2> \) are two TTS and \( L \) is a set of (observable) labels. The product of two TTS over \( L \), denoted \( K_1 \parallel_L K_2 \), is the TTS \( ([S_1 \times S_2], [s_1', s_2'], \alpha_1 \alpha_2) \rightarrow \) such that \( \rightarrow \) is the smallest relation obeying the following rules:

\[
\begin{align*}
\alpha_1 s_1 \xrightarrow{\alpha_1} s_1' & \quad \alpha_2 s_2 \xrightarrow{\alpha_2} s_2' \\
(s_1, s_2) \xrightarrow{\epsilon} (s_1', s_2') & \quad s_1 \xrightarrow{\alpha_1} s_1' \quad s_2 \xrightarrow{\alpha_2} s_2' \\
\left(s_1, s_2\right) \xrightarrow{\alpha_1} (s_1', s_2') & \quad \alpha \in \mathbb{Q}_{\geq 0} \cup L
\end{align*}
\]

It is the case that the “common observations” in \( N_1 \) and \( N_2 \) (relative to \( L \)) are exactly the observations in the TTS product \( [N_1] \parallel_L [N_2] \). This has a direct application when we try to find a critical pair in a TPN \( N \), since it amounts to finding an observation in \([N]\) \parallel_L \([N]\) where the first component had an occurrence of a fault and not the second.

Theorem 1. There is an execution \( \sigma \) in \( K_1 \parallel_L K_2 \) if and only if there are two executions, \( \sigma_1 \) in \( K_1 \) and \( \sigma_2 \) in \( K_2 \), with the same observations: \( \text{obs}_L(\sigma) \equiv \text{obs}_L(\sigma_1) \equiv \text{obs}_L(\sigma_2) \).
null
when we reach a state where place $F$ is marked one (either when we fire a transition or when we pop a node from the stack of visited nodes). We report a counter-example for a pattern $\varphi_D$ as soon as we find one. We also mark nodes that can lead to a place $F$ to a critical pair.

We remark that it is not necessary to fire transition $f.2$ to check property $\varphi_D$, since every execution that contains $f.2$ satisfies $\varnothing(f.2 \lor \text{dead})$. Then, assuming we never fire $f.2$, a counter example to property $\varnothing(f.2 \lor \text{dead})$ corresponds to a non-trivial Strongly Connected Component (SCC). Based on this idea, we propose an adaptation of Tarjan’s SCC Algorithm [20] to “mark” states leading to a cycle (avoiding $f.2$) in the state class graph of $N.1 \times_L N.2$.

Our algorithm generates new nodes (state classes) following a depth-first search order; computes SCC; and mark nodes incrementally. We mark all the nodes in a cycle as soon as we find one. We also mark nodes that can lead to a marked one (either when we fire a transition or when we pop the stuck of visited nodes). We report a counter example for diagnosability as soon as we either: (1) reach a marked node by firing $f.1$; or (2) when we find a SCC containing $f.1$. The net is diagnosable if we finish exploring all the classes. Like with Tarjan’s algorithm, the time complexity is linear in the size of the SCG for $N.1 \times_L N.2$.

A. On-the-Fly Algorithm for Diagnosability

We have improved on the indirect method of Th 3 by defining a dedicated decision algorithm for diagnosability that is on-the-fly (it avoids computing the whole state space of the system when not necessary) and that has better memory usage. When the system is not diagnosable, we return a counter-example that is a trace in $N.1 \times_L N.2$ corresponding to a critical pair.

We also want to make sure that a pattern does not interfere with the system it interacts with. For example, it should not prevent some executions of the system. To this end, we impose three well-formedness conditions on patterns:

1) patterns are total; they should always allow transitions on the labels in $F$, at any time (they never block or delay a transition);
2) patterns are deterministic (the same observations should lead to the same states);
3) labels in $F$ are unobservable ($F \cap L = \emptyset$).

Constraints (1) and (2) can be expressed as a property over all states in $[M]$, namely $\forall s \in [M], a \in F, \theta \in Q_{\geq 0}, \exists t \in T, s', s'' \in [M], (L(t) = a \land s \xrightarrow{a} s' \xrightarrow{a} s'')$.

By analogy with our previous definition of diagnosability, we say that pattern $M$ is diagnosable if it is not possible to find a (critical) pair of executions such that $M$ is found in one but never in the other. We can again reduce this question to a model-checking problem on a twin-plant; this time on the product of the system with the pattern.

Theorem 4. Given a well-formed pattern $M$, with labels $F$, the net $N$ is diagnosable for pattern $M$ if and only if all the maximal executions of the product $(N.1 \times_F M.1) \times_L (N.2 \times_F M.2)$ satisfy $\varnothing(\text{found}.1) \Rightarrow \varnothing(\text{found}.2 \lor \text{dead})$.

Proof. By Th. 1, since $M$ is total, the pattern is detected for the execution $\sigma$ of $N$ iff there is an (equivalent) execution $\sigma_M$ in $[N]\parallel_F [M]$ and property $\varnothing(\text{found})$ is valid for $\sigma_M$. Since $M$ is deterministic, $\sigma_M$ is unique, so it is not possible to find another execution, compatible with $\sigma$, where $\text{found}$ is not marked. Hence we are left with checking the diagnosability of the event $\text{found}$ in the net $N \times_F M$. The result follows from Th. 3 and the fact that $F \cap L = \emptyset$.

We can relax some of the well-formed constraints in the proof of Theorem 4. For instance, we can replace “deterministic” with the weaker property that “detection is unambiguous”: it is not possible to find an execution in $M$ that leads to two markings, one where $\text{found}$ is marked, the

Fig. 2: pattern for “three consecutive $b$ without $f$”

M. For instance, the pattern in Fig. 2 detects executions that have three consecutive occurrence of $b$ without any $f$ in-between. This can be inferred from the marking graph of the pattern (see Fig. 3).

Instead of defining a pattern as a regular language [2], or as a set of timed sequences [21], [22], we use a (prefix-closed) set of executions in $[M]$. Hence we restrict ourselves to time regular languages, meaning sets of executions that can be “realized” with a TPN. This is enough to model every regular set of (untimed) traces.

B. Pattern Diagnosability

Our method can be naturally extended to check for the diagnosability of “patterns of unobservable events” [2]. In our case, a pattern $M$ is a special instance of TPN. We denote $F$ the set of labels occurring in $M$ and we distinguish a place in $M$, say $\text{found}$, that is a witness for detection.

We say that pattern $M$ detects the execution $\sigma$ if $\text{obs}_F(\sigma)$, the $F$-observation of $\sigma$, is an execution of $M$ that “marks” place $\text{found}$; we can run the execution $\text{obs}_F(\sigma)$ in $[M]$ and it satisfies the linear property $\varnothing(\text{found})$.

More generally, we say that pattern $M$ is detected in $N$ when we reach a state where place $\text{found}$ is marked in $N \times_F$.
other not. Nonetheless, this presentation has some merits. For instance each condition can be checked automatically on the marking graph of \( M \) when the net is bounded and has no timing constraints.

V. IMPLEMENTATION AND EXPERIMENTAL RESULTS
We have extended our tool Twina with support for the declaration of product relations and for the construction of “twin machines”. See https://projects.laas.fr/twina/ for installing the tool and for information necessary to reproduce our experiments.

Twina provides an option, \(-twin\), to build the state class graph for the “twin-product” of a net given a fault label. The result can be used as input of the Selt model-checker, which is part of the Tina toolbox. We also provide an option, \(-diag\), for testing diagnosability of single faults using our on-the-fly algorithm of Sect. IV-A. When the system is not diagnosable, it is possible to print a counter-example using the verbose output mode, option \(-v\).

We provide several use cases on the tool website. We only report some of our results here due to space limitations.

One of our biggest model is a version of the WODES diagnosis benchmark of Giua [23], also used in [21], with the addition of timing constraints. This instance is not diagnosable. Diagnosability with the (indirect) approach from Th 3 takes about one hour of computation and generates almost 420 million classes. We return a result in about 5 s with option \(-diag\) (using the same computer) after computing less than 200 000 classes. The counter-example found is of length 63. We are also able to check the diagnosability of the examples taken from [15], [21] and obtain the same verdicts.

To check the diagnosability of a net, \( N \), with respect to a \( F \)-pattern \( M \), it is enough to compute the SCG for a system build from the “twin-plant” product of \( N \times_F M \) and then use a model-checker to check property \( \phi_D \) of Th. 4. Among other benchmarks, we have experimented this approach with an instance of the pattern in Fig. 2, on a timed version of the product transportation system found in [21]. The untimed system is not diagnosable (and it state graph has 14270 markings). By varying the timing constraints, we can switch from a diagnosable behavior to an un-diagnosable one. The size of the resulting SCG vary from about 20000 classes to almost 130000 when the system is not diagnosable. In all these examples, computation time is less than a few seconds.

VI. CONCLUSION
We propose a constructive and unified approach for deciding the diagnosability of single failures and patterns in TPN. Our presentation emphasizes the connections between these two properties. For future work, we hope to benefit from this relationship in order to study more elaborate observability properties; for example a notion of opacity for patterns.

As such, our approach can be naturally extended to the diagnosability of timed pattern. Unfortunately, it was not possible to delve into this question due to space limitations. In this context, proving that the pattern is total and deterministic may be complex; and may require to consider priorities between transitions. Our approach could also be adapted to check for \( k \)-diagnosability (by limiting the number of observable events after a fault) or for \( \Delta \)-diagnosability, by composing our twin-plant model with an “observer” that triggers an alarm if it is possible to let a delay \( \Delta \) elapse after the first occurrence of a fault. We plan to study these questions in a future work.

REFERENCES