COMPUTING
SCIENCE

Predictability Verification with Parallel LTL-X Model Checking Based on Petri Net Unfoldings

Agnes Madalinski and Victor Khomenko

TECHNICAL REPORT SERIES

No. CS-TR-1276       September 2011
Predictability Verification with Parallel LTL-X Model Checking Based on Petri Net Unfoldings

A. Madalinski, V. Khomenko

Abstract

We show that the predictability problem for a Petri net can be reduced to LTL-X model checking. The advantage of this is that existing efficient methods and tools can be employed, in particular parallel model checking based on Petri net unfoldings. The experimental results show that this approach is efficient, and a good level of parallelisation can be achieved.
Bibliographical details

MADALINSKI. A., KHOMENKO, V.

Predictability Verification with Parallel LTL-X Model Checking Based on Petri Net Unfoldings
[By] A. Madalinski, V. Khomenko
(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1276)

Added entries

NEWCASTLE UNIVERSITY

Abstract

We show that the predictability problem for a Petri net can be reduced to LTL-X model checking. The advantage of this is that existing efficient methods and tools can be employed, in particular parallel model checking based on Petri net unfoldings. The experimental results show that this approach is efficient, and a good level of parallelisation can be achieved.

About the authors

Agnes Madalinski obtained an MSc in Parallel and Scientific Computation from the University of Liverpool, UK, in 2000, and a PhD in Electronic and Computer Engineering in 2005 from the University of Newcastle upon Tyne, UK. She completed two post-docs in France, in DistribCom Research Group at INRIA Rennes and in Gemo Team at INRIA Saclay. Since 2008 she is an associate professor in the Faculty of Science and Engineering at the University of Austral de Chile, Valdivia, Chile. She is interested in verification of system with Petri net unfoldings applied to fault diagnosis and asynchronous (self-timed) circuits.

Victor Khomenko obtained his MSc with distinction in Computer Science, Applied Mathematics and Teaching of Mathematics and Computer Science in 1998 from Kiev Taras Shevchenko University, and PhD in Computing Science in 2003 from Newcastle University. He was a Program Committee Chair for the International Conference on Application of Concurrency to System Design (ACSD'10). He also organised the Workshop on UnFOlding and partial order techniques (UFO'07) and Workshop on BALSA Re-Synthesis (RESYN'09). In January 2005 Victor became a Lecturer in the School of Computing Science, Newcastle University, and in September 2005 he obtained a Royal Academy of Engineering / EPSRC Post-doctoral Research Fellowship and worked on the Design and Verification of Asynchronous Circuits (DAVAC) project. After the end of this award, in September 2010, he switched back to Lectureship. Victor’s research interests include model checking of Petri nets, Petri net unfolding techniques, verification and synthesis of self-timed (asynchronous) circuits.

Suggested keywords

PREDICTABILITY
FAULT DIAGNOSIS
PETRI NET UNFOLDINGS
PARALLEL LTL-X MODEL CHECKING
Predictability Verification with Parallel LTL-X Model Checking Based on Petri Net Unfoldings

Agnes Madalinski¹ and Victor Khomenko²
¹Faculty of Engineering Science, University Austral de Chile, Valdivia, Chile
²School of Computing Science, Newcastle University, Newcastle upon Tyne, UK

Abstract

We show that the predictability problem for a Petri net can be reduced to LTL-X model checking. The advantage of this is that existing efficient methods and tools can be employed, in particular parallel model checking based on Petri net unfoldings. The experimental results show that this approach is efficient, and a good level of parallelisation can be achieved.

Keywords: Predictability, Fault diagnosis, Petri net unfoldings, parallel LTL-X model checking.

1. Introduction

Fault diagnosis consists in detecting abnormal behaviours of a physical system. Within the fault diagnosis framework, predictability is a property describing the possibility of predicting a fault before it actually occurs by monitoring the observable behaviour of the system. Predictability implies diagnosability — an important property that determines the possibility of detecting faults by monitoring the observable behaviour. The difference is that diagnosability ensures that the fault can be eventually detected, maybe long time after its occurrence, while predictability allows to detect the fault before it actually occur. Predictability makes it possible to react before the fault causes the system to malfunction, e.g. by issuing a warning or taking some preventing measures.

Predictability has been introduced in [5]. It is based on the seminal work [15], which presents a formal language framework for diagnosis and analysis of diagnosability properties of discrete event systems represented by finite automata. The system’s actions are partitioned into observable and unobservable; furthermore, some of the unobservable actions are designated as faults. The proposed method for diagnosability verification was based on the construction of a diagnoser — an automaton with only observable transitions which allows one to estimate the current state of the system by observing its visible actions. Improvements based on the twin plant method have been introduced in [8, 17], where the basic idea was to build a verifier by constructing the synchronous product of the system with itself on observable transitions. Then, violations of diagnosability can be detected by inspecting execution of the verifier. The diagnoser and verifier approaches have been also used to verify the predictability property in [5] and [6], respectively.

Naturally, the state-based twin plant method suffers from the combinatorial state space explosion problem. That is, even a relatively small system specification can (and often does) yield a very large state space. To alleviate this problem Petri net (PN) unfolding techniques appear promising. The system is modelled as a PN, where each transitions is labelled with the performed action. A finite and complete prefix of a PN unfolding gives a compact representation of all reachable markings of this PN. Executions are considered as partially ordered sets of transitions rather than sequences, which often results in memory savings. Since the introduction of the unfolding technique in [11], it was improved [2], parallelised [7], and applied to various practical applications such as distributed diagnosis [3] and LTL-X model checking [1]. Also, the problem of diagnosability verification based on the twin plant method has been studied in [10] in the context of parallel LTL-X model checking based on PN unfoldings.

This paper adapts the twin plant method deployed in [10] (where an existing parallel LTL-X model checker P UNF [13, 16] based on PN unfoldings is applied) to predictability verification. A verifier is built in such a way that it can produce a witness of non-predictability if it exist. Then, predictability can be expressed as an LTL-X property of the verifier, and to model-check it, a synchronised net is constructed from the verifier and an appropriate Büchi automaton. Finally, the unfolding-based LTL-X model checking [1, 16] is applied to carry out the verification. Experiments show that the proposed approach is quite efficient, and good parallelisation can be achieved.
2. Basic notions

Petri nets. A Petri net is a quadruple \( N = (P, T, F, M_N) \) such that \( P \) and \( T \) are disjoint sets of places and transitions, respectively, \( F \subseteq (P \times T) \cup (T \times P) \) is a flow relation, and \( M_N \) is the initial marking, where a marking is a multiset of places, i.e., a function \( M : P \rightarrow \mathbb{N} = \{0, 1, 2, \ldots \} \) assigning a number of tokens to each place. We adopt the standard rules about drawing nets, viz. places are represented as circles, transitions as boxes, the flow relation by arcs, and the marking is shown by placing tokens within circles. As usual, \( \bullet \mathcal{L} = \{ y \mid (y, z) \in F \} \) and \( \bullet \mathcal{S} = \{ y \mid (z, y) \in F \} \) denote the pre- and postset of \( z \in P \cup T \). In this paper, the presets of transitions are restricted to be non-empty, i.e., \( \bullet \mathcal{S} \neq \emptyset \) for every \( t \in T \).

A transition \( t \in T \) is enabled at a marking \( M \), denoted \( M[t] \), if for every \( p \in \bullet \mathcal{S}, M(p) \geq 1 \). Such a transition can fire, leading to a marking \( M' \equiv M - \bullet \mathcal{S} + \bullet \mathcal{L} \), where ‘−’ and ‘+’ stand for the multiset difference and sum, respectively. We denote this by \( M[t]M' \). For a finite sequence of transitions \( \sigma = t_1 \ldots t_k (k \geq 0) \), we write \( M[\sigma]M' \) if there are markings \( M_1, \ldots, M_k+1 \) such that \( M_1 = M, M_{k+1} = M' \) and \( M[t_i]M_{i+1} \) for \( i = 1, \ldots, k \). If \( M \equiv M_N \), we call \( \sigma \) an execution of \( N \). Analogously, infinite executions can be defined.

The set of reachable markings of \( N \) is the smallest (w.r.t. \( \subseteq \)) set \( [M_N] \) containing \( M_N \) and such that if \( M \in [M_N] \) and \( M[t]M' \) for some \( t \in T \) then \( M' \in [M_N] \). A PN \( N \) is \( k \)-bounded if, for every reachable marking \( M \) and every place \( p \in P, M(p) \leq k \), and safe if it is \( 1 \)-bounded. In what follows, we assume that the PNs we deal with are safe. A marking of \( N \) is called a deadlock if it enables no transitions. \( N \) is deadlock-free if none of its reachable markings is a deadlock.

A labelled PN \( \bar{N} = (N, O, U, \ell) \) extends a PN \( N \) with disjoint sets \( O \) and \( U \) of observable and unobservable transition labels and a labelling function \( \ell : T \rightarrow O \cup U \). \( \bar{N} \) inherits the operational semantics of the underlying net \( N \). We lift the notion of enabledness and firing to transition labels: \( M[\ell(t)]M' \) if \( M[t]M' \). Moreover, the domain of \( \ell \) can be extended to finite and infinite sequences of transitions in a natural way, and \( \ell(\sigma) \) is called a trace of \( N \) if \( M[\sigma] \), where \( \sigma \) is a finite or infinite execution. \( \bar{N} \) is divergence-free if none of its reachable markings enables an infinite trace comprised of unobservable actions. For a (finite or infinite) trace \( \xi \) we denote by \( Obs(\xi) \) the projection of \( \xi \) onto \( O \).

Predictability. The predictability problem is formulated on a labelled Petri net \( \bar{N} \) that is assumed to be deadlock-free and divergence-free. Intuitively, its observable actions correspond to controller commands and sensor readings, while the unobservable ones correspond to some internal activity that is not recorded by sensors. Some actions are designated as faults, and, w.l.o.g., the set of faults \( F \subseteq U \) (for predictability to hold, one should be able to predict faults before they occur, and so making fault transitions observable does not affect this property). As an example, consider Fig. 1, where \( O = \{a, b, c\}, U = \{u, f\} \) and \( F = \{f\} \).

![Figure 1: An example net \( \bar{N} \).](image-url)

The following definition of predictability is based on the one in [6]. (We have simplified it somewhat; the given definition coincides with that in [6] on finite state systems.) Intuitively, a system is predictable w.r.t. a fault \( f \) if it does not have a pair of traces whose initial parts have the same projection onto the observable actions and do not contain occurrences of \( f \), and the continuation (which can be finite or infinite) of one of these traces starts with \( f \) while the continuation of the other is infinite and does not contain occurrences of \( f \). For example, the system in Fig. 1 is not predictable, as it has a pair of traces \( (a, (aub)^n) \) whose initial parts have the same projection \( a \) onto the observable actions, the first trace is continued with \( f \), and the second one is infinite and does not contain occurrences of \( f \); hence, after the first occurrence of \( a \), it is not possible to predict whether the fault will occur or not.

Formally, a deadlock-free and divergence-free labelled Petri net \( \bar{N} \) is predictable w.r.t. a fault \( f \) if it does not have a pair of traces of the form \((\xi_1, f \xi_2, \xi_1' f \xi_2')\), where:

1. \( \xi_1 \) and \( \xi_1' \) are finite traces containing no occurrences of \( f \) and satisfying \( Obs(\xi_1) = Obs(\xi_1') \); and
2. \( \xi_2 \) and \( \xi_2' \) contain no occurrences of \( f \); and
3. \( \xi_1' \) is infinite.

A pair of traces satisfying the above conditions constitutes a witness of predictability violation, see Fig 2. Such a witness is returned by our unfolding based LTL-X model checking approach in case the property does not hold, and it can be used for debugging.

Note that some non-essential choices have been made in the above definition. First, one could allow \( \xi_2 \)
to contain further occurrences of \( f \); however, any such witness can be converted to the form required by the above definition by truncating the first trace so that it contains a single occurrence of \( f \). Second, no restriction is placed on the length of \( \varphi \) — it can be infinite, finite or even empty. In fact, it is possible to simplify the definition by requiring that \( \varphi \) is empty. It is easy to show that any combination of these choices yields the same class of systems. The actual choices that have been made were motivated by technical convenience — the witnesses of this form is exactly the one returned by our method.

**LTL-X and Büchi automata.** There are two orthogonal views on system computation. According to the state-based view, a computation is a potentially infinite sequence of states \( s_0 s_1 s_2 \ldots \), such that for each \( i \), \( s_{i+1} \) is reachable from \( s_i \) in one step. According to the action-based view, a computation is a potentially infinite sequence of actions \( a_0 a_1 a_2 \ldots \) performed by the system.

**Linear time temporal logic (LTL)** [12] is a logic allowing to specify the properties of computations. LTL is built up from: (i) a set \( AP \) of atomic propositions; (ii) the usual Boolean connectives \( \neg, \land \) and \( \lor \); and (iii) the temporal modalities \( \bigcirc \) (next-state) and \( U \) (until).

In the case of state-based computations the atomic propositions correspond to state predicates, e.g. for safe PN’s they are usually chosen as follows: for each place \( p_i \), the corresponding atomic proposition, also denoted by \( p_i \), is true for a computation \( s_0 s_1 s_2 \ldots \) iff at state \( s_0 \) place \( p_i \) contains a token. In the action-based case, the atomic propositions usually corresponds to the actions of the system, e.g. the atomic proposition \( a \) is true for a computation \( a_0 a_1 a_2 \ldots \) iff \( a_0 = a \).

In this paper only the derived modality \( \eventually \) (that can be defined via \( U \)) will be needed: \( \eventually \varphi \) is true iff there exists \( i \) such that \( \varphi \) is true for the computation \( s_0 s_{i+1} s_{i+2} \ldots \) (and similarly for the action-based case).

The logic LTL-X is the \( \bigcirc \)-free fragment of LTL. LTL-X plays a very prominent role in formal verification. In fact, Lamport has famously argued that every ‘sensible’ LTL specification must be expressible without the \( \bigcirc \) operator [9].

A Büchi automaton is an extension of a non-deterministic finite state automaton to infinite inputs. It accepts an infinite input sequence iff some corresponding execution visits any of the designated final states infinitely many times. The language of a Büchi automaton is defined as the set of all infinite inputs that can be accepted.

The following technique is often used to formally verify whether a system \( S \) satisfies an LTL property \( \varphi \) [18]. Deciding whether all computations of \( S \) satisfy \( \varphi \) is equivalent to deciding whether some computation of \( S \) satisfies \( \neg \varphi \). To complete the latter task, \( \neg \varphi \) is converted into a Büchi automaton \( A - \varphi \) accepting the computations satisfying \( \neg \varphi \) [4]. Then, \( S \) and \( A - \varphi \) are synchronised in such a way that the language of the resulting Büchi automaton \( S \times A - \varphi \) is the intersection of the language of \( A - \varphi \) and the set of all the possible computations of \( S \). Hence, in this way one can reduce the original verification problem to checking if the language accepted by the Büchi automaton \( S \times A - \varphi \) is empty, which is the case iff no final state is both reachable from the initial state and lies on a cycle.

**Unfolding prefixes.** The unfolding of a PN \( N \) is a (potentially infinite) acyclic net that can be obtained by starting from the initial marking of \( N \) and successively firing its transitions, as follows: (a) for each new firing a fresh transition (called an event) is generated; (b) for each newly produced token a fresh place (called a condition) is generated.

Due to its structural properties (such as acyclicity), the reachable markings of \( N \) can be represented using configurations of the unfolding. A configuration \( \kappa \) is a downward-closed set of events (it means that if \( e \in \kappa \) and \( f \) is a causal predecessor of \( e \), then \( f \in \kappa \)) without choices (i.e. for all distinct events \( e, f \in \kappa, e \cap f = \emptyset \)). Intuitively, a configuration is a partially ordered execution, i.e. an execution where the order of firing of concurrent events is not important. The local configuration of an event \( e \), denoted by \([e]\), is the smallest (w.r.t. set inclusion) configuration containing \( e \) (it consists of \( e \) and its causal predecessors); \( Mark(\kappa) \) denotes the marking of \( N \) reached by any execution corresponding to the events in \( \kappa \) (note that there can be several such executions, but they only differ by the order of firing of
was implemented in PUNF tool [13]. Hence, though in this paper we are mostly concerned with predictability of systems modelled as low-level PNs, and our benchmarks are low-level PNs, all the results can be trivially generalised to high-level PNs.

3. Predictability via LTL-X verification

In this section we show how checking predictability can be reduced to LTL-X model checking. Then, the unfolding based method outlined above can be used to solve the problem.

**Building a verifier.** A verifier \( V \) is a labelled Petri net that is obtained as a hybrid product (defined later) of the original system \( N \) with itself. The key property of \( V \) is that whenever it has an infinite trace containing an occurrence of \( f \), the projection of this execution onto the pair of composed nets yields a pair of traces constituting a witness of predictability violation, and vice versa, whenever such a witness exists, the corresponding pair of traces can be converted into an infinite trace of \( V \) containing an occurrence of \( f \). Hence, \( N \) is not predictable iff the corresponding \( V \) has an infinite trace containing \( f \), and the latter can be expressed as a simple LTL-X property of \( V \). We now explain the construction of \( V \) and prove its relevant properties.

Recall that a witness of predictability violation is comprised of two traces of \( N \), which are initially synchronised on observable action, and then become desynchronised when \( f \) occurs, cf. Fig. 2. In order to construct \( V \), two replicas of \( N \) are taken (to reason about a pair of traces); we will denote these two nets by \( N^1 \) and \( N^2 \). Since \( f \) is not supposed to fire in the second trace of the witness, the transitions labelled with \( f \) are removed from \( N^2 \). Then \( N^1 \) and \( N^2 \) are synchronised using the following hybrid construction, which combines the properties of the usual synchronous product and the interleaving composition (the latter simply places the nets being composed side-by-side, forming thus one net whose parts do not interact with each other).

First, we follow the usual synchronous product construction. Intuitively, \( N^1 \) and \( N^2 \) are put side-by-side, and then each observable transition in \( N^1 \) is fused with each transition in \( N^2 \) that has the same label (each fusion produces a new transition, inheriting the common label); however, in contrast to the synchronous product construction, which removes the original visible transitions, we remove only the ones in \( N^1 \), while preserving those in \( N^2 \). Fig. 3(a) shows the result of this step for the example net in Fig. 1. The superscript is used to distinguish nodes belonging to \( N^1 \) from those belonging to \( N^2 \), e.g. there are two copies of \( u \) in \( V \).
of two mutually exclusive places, \(\mathcal{N}\) fires in transitions in output transitions correspond to the removed observable scripts — they are considered ‘common’. The greyed u for the example net in Fig. 1.

Figure 3: The hybrid product construction of a verifier

(a) the first step of the hybrid product construction: two replicas of \(\mathcal{N}\) (with the fault \(f\) removed from the second replica) are put side-by-side, their observable transitions having the same label are fused, and the original observable transitions in the first replica removed.

(b) the second step of the hybrid product construction, yielding the verifier \(V\): places \(p_f\) and \(\overline{p_f}\) are added and connected to \(f\), and \(p_f\) is connected by read arcs to the observable transitions of second replica; optionally, \(\overline{p_f}\) is connected by read arcs to the fusion transitions.

\(u^1\) and \(u^2\); the fusion transitions do not have superscripts — they are considered ‘common’. The greyed out transitions correspond to the removed observable transitions in \(\mathcal{N}^1\).

\(\mathcal{N}^1\) and \(\mathcal{N}^2\) should be de-synchronised as soon as \(f\) fires in \(\mathcal{N}^1\). This is implemented by a switch consisting of two mutually exclusive places, \(\overline{p_f}\) and \(p_f\), as illustrated in Fig. 3(b) in bold (if there are several transitions labelled by \(f\) in \(\mathcal{N}\), these places are shared by all such transitions). Initially, the place \(\overline{p_f}\) is marked and \(p_f\) is not, indicating that the two nets should synchronise on observable actions — the absence of a token on \(p_f\) prevents the instances of the observable transitions belonging to the second net from firing, due to the read arcs between \(p_f\) and these instances. However, once \(f\) has fired, the token moves from \(\overline{p_f}\) to \(p_f\), enabling thus the visible transitions belonging to \(\mathcal{N}^2\), and allowing \(\mathcal{N}^2\) to run unrestrictedly, without synchronising with \(\mathcal{N}^1\).

Note that though the fusion transitions are not disabled by firing of \(f\) and so synchronisation remains pos-sible, this synchronisation is no longer required, and \(\mathcal{N}^2\) can run unrestrictedly; moreover, firing a fusion transition has the same effect on the marking of \(\mathcal{N}^2\) as firing the corresponding original visible transition. Alternatively, one can forbid firing the fusion transitions after an occurrence of \(f\), by adding read arcs between \(p_f\) and them, which are shown as dashed lines in Fig. 3b.

On the other hand, since after an occurrence of \(f\) the behaviour of \(\mathcal{N}^1\) does not matter (it will not contain any further occurrences of \(f\) as \(\overline{p_f}\) becomes empty and will never contain a token again), the original observable transitions of \(\mathcal{N}^1\) are not needed (and are removed by our construction).

Having defined the verifier \(V\), we establish the correspondence between its executions and pairs of executions of \(\mathcal{N}\). The idea is to project any execution of \(V\), transition-by-transition, as follows:

- if the fired transition is not a fusion transition and belongs to \(\mathcal{N}^1\) (resp. \(\mathcal{N}^2\)) then the corresponding transition of \(\mathcal{N}\) is appended to the first (resp. second) element of the pair;

- if the fired transition is a fusion transition obtained by fusing transition \(t^1\) of \(\mathcal{N}^1\) and \(t^2\) of \(\mathcal{N}^2\) then \(t\) and \(t'\) are appended to the first and second elements of the pair, respectively.

Obviously, by applying the labelling function \(\ell\), this correspondence can be lifted to traces. Furthermore, it is easy to see that any pair of traces of \(\mathcal{N}\) constituting a witness of predictability violation can be converted into an infinite trace of \(V\) containing an occurrence of \(f\).

There is a subtle point in this construction that is nevertheless important for the correctness of the proposed method: one has to ensure that whenever the trace of the verifier is infinite and contains an occurrence of \(f\), the second element of the pair of projected traces is infinite (i.e. the situation when the first element is infinite while the second is finite is impossible). Fortunately, this is always the case due to the assumption that \(\mathcal{N}\) is divergence-free. Indeed, if the first element of the projection is infinite, \(\mathcal{N}^1\) must have executed infinitely many fusion transitions (as other visible transitions are removed from \(\mathcal{N}^1\) by the hybrid product construction, and it is possible to execute only finitely many invisible transitions before firing a visible one due to the divergence-freeness assumption). Hence, the second projection must be also infinite.

**LTL-X model checking for non-predictability.** As explained above, given the verifier \(V\), checking the complement \(\overline{pred}\) of predictability property can be reduced to checking the existence of an infinite trace of \(V\).
containing an occurrence of f (note that it is not necessary to express the predictability property itself, as the Büchi automaton is built for the complement of the property). In state-based LTL-X it can be expressed as

\[ \text{pred} \equiv \Diamond p_f, \]

(recall that \( p_f \) is one of the two ‘switch’ places in \( \mathcal{V} \)).

The Büchi automaton \( \mathcal{A}_{\text{pred}} \) for \( \text{pred} \) and the corresponding Büchi net \( \mathcal{N}_{\text{pred}} \) are shown in Fig. 4. This net is then synchronised with the verifier \( \mathcal{V} \), and we will denote the result by \( \mathcal{V}_{\text{sync}} \). Note that the kind of synchronisation proposed in [1] is non-standard, and is aimed at preserving as much concurrency as possible in order to keep the unfolding prefix small. This is achieved by synchronising the net with the Büchi net only on transitions that add or remove tokens from places which appear in the LTL-X formula (in Fig. 3(b), these are \( f^1 \)), see [1] for more detail. \( \mathcal{V}_{\text{sync}} \) can be fed to \( \text{PUNF} \) unfolder [16], which finds an infinite trace of \( \mathcal{V} \) satisfying the \( \text{pred} \) property \((a f^1 (u^2 b^3 a^2)^\omega)\) in the running example, if there is one.

4. Experimental results

In order to test the efficiency of the proposed approach to predictability verification, we constructed several series of scalable benchmarks, by modifying those coming from the area of diagnosability in such a way as to make them interesting from the predictability point of view. They are explained below.

**PCConcSame\((n)\) and PCConcDiff\((n)\)** These benchmarks series (see Fig. 5(top)) were inspired by a similar series in [14]. They model a producer-consumer system with an \( n \)-slot buffer, where the slots can be accessed concurrently, i.e. the producer, after producing an item (transition \( \text{produce} \)), non-deterministically chooses an empty slot to deposit the produced item (transitions \( \text{deposit} \)), and the consumer non-deterministically chooses a non-empty slot to take an item from (transitions \( \text{take}_c \)) and then consumes it (transition \( \text{consume} \)). However, the location of the fault transition has been changed from that in [14], to make the fault interesting from the point of view of predictability. The fault transition \( f \) is enabled when all the buffer slots are full (note the read arcs between \( f \) and the places \( \text{full}_i \)) and the producer is ready to deposit a new item into the buffer (we call the system’s states satisfying these properties **critical**). In a critical state this new item can be lost, which is modelled by the fault transition \( f \). In this system, the only observable transitions are produce and take\(_c\), and the only difference between the \( \text{PCConcSame}\((n)\) and \( \text{PCConcDiff}\((n)\) series is that in the former transitions take\(_c\) have the same observable label, while in the latter they have different observable labels.

**PCSeq\((n)\)** This benchmarks series (see Fig. 5(bottom)) is similar to the ones described above, but the buffer is sequential. The producer deposits newly produced items into the first slot, items propagate from slot to slot in a pipeline manner, and the consumer takes items from the last slot. The semantics of the fault transition is the same (if all the slots are full and the producer is ready to deposit a new item, i.e. the system is in a critical state, then this item can be lost), and the only observable transitions are produce and take\(_c\).

These benchmarks series are not predictable. Intuitively, whenever the system is in a critical state, two scenarios are possible — either to fire \( f \), or to consume an item from the buffer, disabling \( f \). Hence it is not possible to know in advance if \( f \) will occur.

In order to test our method on predictable benchmarks, we constructed the predictable versions of these
Table 1: Benchmarks statistics.

benchmark series. The key idea is to eliminate the choices between $f$ and the transitions taking an item from the buffer (take, in the former series and take in the latter one), by giving priority to $f$ (intuitively, $f$ is considered ‘faster’ than any such transition).

In the PCConCSame($n$) and PCConCDiff($n$) benchmarks the predictability is achieved by replicating each transition take, so that there are $n$ replicas take$_{ij}$ of it. Each take$_{ij}$ inherits the label of the original transition take, takes tokens from ready_to_take and full.
and puts tokens on \texttt{ready.to.consume} and empty}_j (just as the original transition \texttt{take}_i did). Moreover, if \( j \neq i \), \texttt{take}_j tests that \( j \)-th slot is empty (by a read arc between \texttt{take}_j and empty}_j), and if \( j = i \) the transition tests that the producer is not ready to deposit a new item into the buffer (by a read arc between \texttt{take}_j and \texttt{ready.to.produce}). Observe that whenever the system is in a critical state (and so \( f \) is enabled) none of \texttt{take}_j can be enabled, and in all other states at least one of \texttt{take}_j is enabled (provided \texttt{ready.to.take} is marked and the buffer is not empty), which makes the system predictable: Intuitively, the observer can deduce whether the system is in a critical state by checking if the number of firings of produce minus the total number firings of the \texttt{take}_j transitions is \( n + 1 \), and all the traces starting at a critical state contain \( f \).

In the PCS\texttt{EO}(n) benchmarks the predictability is achieved in a similar way, by replicating transition \texttt{take}. Each replica \texttt{take}_i inherits the label of the original transition \texttt{take}, takes tokens from \texttt{ready.to.take} and full}_i and puts tokens on \texttt{ready.to.consume} and empty}_n (just as the original transition \texttt{take}_i did). Moreover, if \( i \neq n \), \texttt{take}_j tests that \( j \)-th slot is empty (by a read arc between \texttt{take}_j and empty}_j), and if \( j = n \) the transition tests that the producer is not ready to deposit a new item into the buffer (by a read arc between \texttt{take}_j and \texttt{ready.to.produce}).

These benchmarks are available from the authors upon request. It should be noted that we experienced some difficulty in finding relevant benchmarks, as it seems so far only theoretical work has been done in the area of predictability, but very few, if any, practical experiments conducted (we did not find any non-trivial publicly available predictability benchmarks). This is also the reason why the results are not compared against other tools — we simply could not find any comparable public domain tools.

The benchmarks statistics is shown in Table 1, where the meaning of the columns is as follows (from left to right): name of the benchmark; the numbers of places, transitions and reachable markings in the original PN; and the numbers of places, transitions and reachable markings in \( \mathcal{V} \); the number of reachable markings in \( \mathcal{V}_{\text{sync}} \). Since the benchmarks have a regular structure, all these numbers can be obtained using the formulae given in the table.

The experimental results are summarised in Table 2, where the meaning of the columns is as follows (from left to right): name of the benchmark; the numbers of conditions and non-cut-off events in the LTL-X tableaux (i.e. unfolding prefix) built for the net obtained by synchronising the corresponding verifier with the Büchi automaton shown in Fig. 4; and the verification runtime. These results were obtained with the help of the unfolding based LTL-X model-checker PUNF [13]. All experiments were conducted on a PC with 64-bit Windows 7 operating system, an Intel Core i7 2.8GHz Processor with 8 cores and 4GB RAM. No parallelisa-

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LTL-X tableaux for ( \mathcal{V}_{\text{sync}} )</th>
<th>Time (sec)</th>
<th>LTL-X tableaux for ( \mathcal{V}_{\text{sync}} )</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#C</td>
<td>#E</td>
<td></td>
<td>#C</td>
</tr>
<tr>
<td>PC\texttt{CONC\texttt{SAME}}(3)</td>
<td>380</td>
<td>148</td>
<td>&lt;1</td>
<td>472</td>
</tr>
<tr>
<td>PC\texttt{CONC\texttt{SAME}}(4)</td>
<td>1201</td>
<td>494</td>
<td>&lt;1</td>
<td>1558</td>
</tr>
<tr>
<td>PC\texttt{CONC\texttt{SAME}}(5)</td>
<td>4213</td>
<td>1773</td>
<td>&lt;1</td>
<td>5528</td>
</tr>
<tr>
<td>PC\texttt{CONC\texttt{SAME}}(6)</td>
<td>15438</td>
<td>6555</td>
<td>&lt;1</td>
<td>20272</td>
</tr>
<tr>
<td>PC\texttt{CONC\texttt{SAME}}(7)</td>
<td>57686</td>
<td>24584</td>
<td>8</td>
<td>75588</td>
</tr>
<tr>
<td>PC\texttt{CONC\texttt{SAME}}(8)</td>
<td>217803</td>
<td>92988</td>
<td>497</td>
<td>284662</td>
</tr>
<tr>
<td>PC\texttt{CONC\texttt{SAME}}(9)</td>
<td>827867</td>
<td>353827</td>
<td>8927</td>
<td>memory overflow</td>
</tr>
<tr>
<td>PC\texttt{CONC\texttt{DIFF}}(3)</td>
<td>394</td>
<td>154</td>
<td>&lt;1</td>
<td>418</td>
</tr>
<tr>
<td>PC\texttt{CONC\texttt{DIFF}}(4)</td>
<td>1197</td>
<td>492</td>
<td>&lt;1</td>
<td>1396</td>
</tr>
<tr>
<td>PC\texttt{CONC\texttt{DIFF}}(5)</td>
<td>4105</td>
<td>1725</td>
<td>&lt;1</td>
<td>5078</td>
</tr>
<tr>
<td>PC\texttt{CONC\texttt{DIFF}}(6)</td>
<td>14864</td>
<td>6301</td>
<td>&lt;1</td>
<td>19012</td>
</tr>
<tr>
<td>PC\texttt{CONC\texttt{DIFF}}(7)</td>
<td>55218</td>
<td>23492</td>
<td>5</td>
<td>72060</td>
</tr>
<tr>
<td>PC\texttt{CONC\texttt{DIFF}}(8)</td>
<td>207829</td>
<td>88572</td>
<td>381</td>
<td>274708</td>
</tr>
<tr>
<td>PC\texttt{CONC\texttt{DIFF}}(9)</td>
<td>788415</td>
<td>336347</td>
<td>6917</td>
<td>1051222</td>
</tr>
<tr>
<td>PC\texttt{SEO}(3)</td>
<td>134</td>
<td>47</td>
<td>&lt;1</td>
<td>236</td>
</tr>
<tr>
<td>PC\texttt{SEO}(4)</td>
<td>182</td>
<td>68</td>
<td>&lt;1</td>
<td>587</td>
</tr>
<tr>
<td>PC\texttt{SEO}(5)</td>
<td>235</td>
<td>91</td>
<td>&lt;1</td>
<td>1808</td>
</tr>
<tr>
<td>PC\texttt{SEO}(6)</td>
<td>301</td>
<td>120</td>
<td>&lt;1</td>
<td>6219</td>
</tr>
<tr>
<td>PC\texttt{SEO}(7)</td>
<td>370</td>
<td>151</td>
<td>&lt;1</td>
<td>22670</td>
</tr>
<tr>
<td>PC\texttt{SEO}(8)</td>
<td>452</td>
<td>189</td>
<td>&lt;1</td>
<td>85177</td>
</tr>
<tr>
<td>PC\texttt{SEO}(9)</td>
<td>537</td>
<td>228</td>
<td>&lt;1</td>
<td>325394</td>
</tr>
</tbody>
</table>

Table 2: Experimental results.

\footnote{There is no need to test the status of \( i \)-th slot by a read arc — \texttt{take}_i can only be enabled if it is full, as it takes a token from full}_i.}
function was used for the results in this table.

In the PCCONCSAME(n) and PCCONCDIFF(n) system, the producer and consumer have concurrent access to n buffer slots, and so there are many arbitrating choices in the resulting behaviour. The unfolding prefixes tend to be large for such systems, and so the corresponding runtimes are much larger than those for the PCSEQ(n) benchmarks. However, this makes it convenient to demonstrate the effect of parallelisation. The experimental results presented in Table 3 demonstrate the effect of deploying several processor cores on the runtime. One can see that good speedups have been achieved for large benchmarks (up to a factor of 1.9 on two cores and 4.5 on eight cores).

5. Conclusions

One can see that the developed method for predictability verification performs quite well, especially on highly concurrent systems like the PCSEQ(n) series. Furthermore, a good level of parallelisation can be achieved. However, the used benchmarks are rather artificial, and so these results should be treated with caution. Larger and more practical benchmarks would allow to draw better conclusions about the performance of the method, but there is a severe lack of public domain benchmarks in the predictability community.

We also note that the proposed approach can be trivially generalised to systems modelled by high-level PNs, as the parallel unfolding-based LTL-X model checking works for them as well [16].

Acknowledgements. This research was supported by the EPSRC grant EP/G037809/1 (VERDAD) and FONDECYT project No11090257.

References