Survey of Petri nets Slicing

Yasir Imtiaz Khan and Nicolas Guelfi

University of Luxembourg, Laboratory of Advanced Software Systems
6, rue R. Coudenhove-Kalergi, Luxembourg
{yasir.khan,nicolas.guelfi}@uni.lu

Abstract. Petri nets slicing is a technique that aims to improve the verification of systems modeled in Petri nets. Different Petri nets slicing constructions are studied along with the algorithms to compute them. Petri nets slicing was first developed to facilitate debugging but then developed for alleviating the state space explosion problem for model checking Petri nets. This article has twofold objectives, the first is to unify all the existing slicing algorithms syntactically by the definition of a standard abstract syntax and rewriting of the studied slicing algorithms. The second is to discuss the contribution of each slicing construction and to compare the major differences between them. One of the interesting exploitation of the survey is for the selection and improvement of slicing techniques for approaches concerned by optimizing verification of state event models.

Key words: Petri nets, Slicing, Model checking, Testing

1 Introduction

Petri nets have been extensively used to model and analyze concurrent and distributed system since their birth. Among several dedicated analysis techniques for Petri nets, model checking and testing are used more commonly. A typical drawback of model checking is its limits with respect to the state space explosion problem: as systems get moderately complex, completely enumerating their states demands a growing amount of resources, which in some cases makes model checking impractical in terms of time and memory consumption. Similarly testing suffers with the problems such as large input amount of test data, test case selection etc. As a result an intense field of research is targeting to optimize these verification techniques, either by reducing the state space or by improving the test input data. A technique called Petri net slicing falls into the first category. However, Petri net slicing (PN slicing) is a syntactic technique used to reduce a Petri net model (PN model) based on the given *criterion*. The given *criterion* refers to the point of interest for which PN model is analyzed. The sliced part constitutes only that part of a PN model that may affect the given *criterion*.

In this article, we review existing PN slicing techniques that can be found in the present iterature [2–9, 12]. We have two objectives, the first is to unify all the existing slicing algorithms syntactically and the second is to discuss the contribution of each work and compare major semantic differences between them.

The remaining part of the paper is structured as follows: in the section 2 , we give an overview and background of slicing together with their specific types. Section 3 consists of formal definitions necessary for the understanding of proposed slicing constructions. In the section 4 and 5, reviewed existing PN slicing techniques that can be found in the present iterature. Details about the underlying theory and techniques for each slicing construction are given. In tthe section 6, a comparative analysis is given for all the existing PN slicing algorithms. In the section 7, we draw the conclusion and discuss the future work regarding the PN slicing.

2 Overview and Background

The term slicing was coined by M.Weiser for the first time in the context of program debugging [13]. According to Wieser proposal a program slice PS is a reduced, executable program that can be obtained from a program P based on the variables of interest and line number by removing statements such that PS replicates part of the behavior of program.

To explain the basic idea of *program slicing* according to Wieser [13], let us consider an example program shown in Fig.1,. Fig.1(a) shows a program which requests a positive integer number n and computes the sum and the product of the first n positive integer numbers. We take as *slicing criterion* a line number and a set of variables, $C = (line10, \{product\})$.

Fig.1(b) shows sliced program that is obtained by tracing backwards possible influences on the variables: In line 7, *product* is multiplied by i, and in line 8, i is incremented too, so we need to keep all the instructions that impact the value of i. As a result all the computations that do not contribute to the final value of *product* have been sliced away (interested reader can find more details about *program slicing* from [11, 14]).

```
(1)  read(n) ;
(2)  i := 1 ;
(3)  sum := 0 ;
(4)  product := 1 ;
(5)  while i <= n do
        begin
(6)        sum := sum + i ;
(7)        product := product * i ;
(8)        i := i + 1 ;
        end ;
(9)  write (sum) ;
(10) write (product) ;
```

(a) Example program.

```
read(n) ;
i := 1 ;

product := 1 ;
while i <= n do
   begin

      product := product * i ;
      i := i + 1 ;
   end ;

write (product) ;
```

(b) Program slice w.r.t. (10,product).

Fig. 1. An example program and sliced program w.r.t. given *criterion*

PN slicing is a technique used to syntactically reduce a PN model in such a way that at best the reduced PN model contains only those parts that may influence the property the PN model is analyzed for. Let us take a simple example of Petri net model to explain the basic idea of PN slicing. In general, it starts by identifying which places or transitions in the PN model are directly concerned by a property. These places constitute the *slicing criterion*. The slicing construction takes all the transitions that create or consume tokens from the criterion places, plus all the places that are pre-condition for those transitions. This step is repeated for later places, until reaching a fix point (see Alg.7). As shown in the Fig.2, a Petri net model is sliced with respect to the place B (a given *slicing criterion*). The sliced part only constitutes the part of the model that is required to analyze the properties concerning to the given place B. The rest of the places and transitions are discarded.

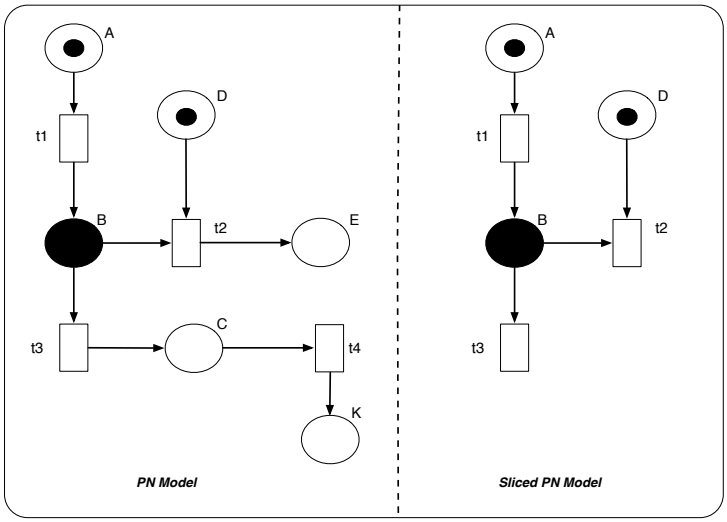

Fig. 2. An example Petri net model and sliced Petri net model w.r.t. given *criterion*

2.1 Types of Slicing

Roughly, we can divide PN slicing in to two major classes (as shown in Fig.4), which are:

- Static Slicing
- Dynamic Slicing

Static Slicing: A slice is said to be static if the initial markings of places are not considered for generating the slice. Only the set of places are considered as a *slicing criterion*. The static slicing starts from the given criterion place and includes all the pre and post set of transitions together with their incoming places. There may exist a sequence of transitions in the resultant slice that is not fireable because some of the pre places of transitions are not initially marked and do not get markings from any other way. Fig.2 is an example of static slice, the slice is generated by considering the criterion place B whereas the initial markings are not used to generate the slice.

An extension to the static slicing can also be used to reduce the slice size. The extension is called condition slicing and the idea is to include a subset of behaviors in the sliced PN model instead of all the behaviors. In addition to the set of places, *Slicing criterion* consists of a sequence of the transitions. The resultant slice obtained by the condition slicing is smaller as compared to the static slicing. The reason for a smaller slice is the inclusion of a particular sequence of transitions around the criterion places. The condition slicing is very useful when analyzing a particular behavior, but limits the scope of verification due to the exclusion of some sequences of transitions.

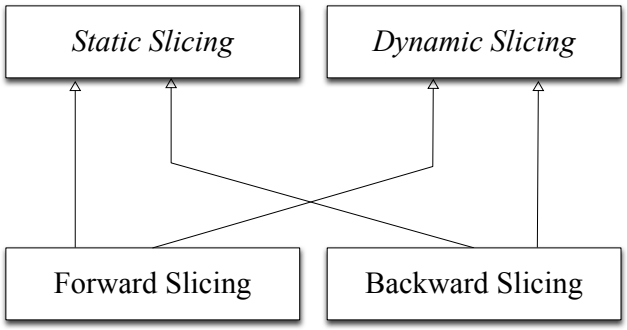

Fig. 3. Different types of PN slicing

Dynamic Slicing: A slice is said to be dynamic if the initial markings of places are considered for generating the slice. The *slicing criterion* will utilize the available information of initial markings and a more smaller slice can be generated. For a given *slicing criterion*, that consist of the initial markings and a set of places for a PN model, we are interested to extract a subnet with those places and transitions of PN model that can contribute to change the marking of a criterion place in any execution starting from the initial marking.

Dynamic slicing can be useful, e.g., in debugging. Consider for instance that the user is analyzing a particular trace for a marked PN model (using a simulation tool) such that an erroneous state is reached. In this case, we are interested

in extracting a set of places and transitions (more formally, a subnet) that may erroneously contribute tokens to the places of interest such that the user can more easily locate the bug.

There are two ways to compute the static and dynamic slices that are forward and backward slicing. A forward slicing starts from the initially marked places and by forward traversal of a PN model untill the criterion places, a slice is generated. Backward slicing starts from the criterion places and then by backward traversal all the incoming transtions together with their input places, slice is obtained.

3 Formal Definitions

In this section, we give the basic definitions needed for providing a unified formal definition framework for the slicing alogorithms considered by our study. To the best of our knowledge existing slicing algorithms are proposed for Petri nets or Algebraic Petri nets. First of all, we shall informally describe Petri nets and Algebraic Petri nets and then we shall give formal definitions of them.

A Petri net is a directed bipartite graph, whose two essential elements are places and transitions. Informally, Petri nets places hold resources (also known as tokens) and transitions are linked to places by input and output arcs, which can be weighted. Usually, a Petri net has a graphical concrete syntax consisting of circles for places, boxes for transitions and arrows to connect the two. The semantics of a Petri net expresses the non-deterministic firing of transitions in the net. Firing a transition means consuming tokens from the set of places linked to the input arcs of the transition and producing tokens into the set of places linked to the output arcs of the transition. Various evolutions of Petri nets have been created, among others in Algebraic Petri nets the level of abstraction of Petri nets is raised by using complex structured data [10]. Algebraic Petri Nets has two aspects, the control aspect, which is handled by a Petri Net and the data aspect, which is handled by one or many algebraic abstract data types (AADTs). (Note: we refer the interested reader to Appendix for the details on algebraic specifications used in the formal definition of APNs for our work.)

Definition 1. ***(Petri net)***
A Petri Net $PN =< P, T, f, \lambda, m_0 >$ *consist of:*
∘ P *and* T *as finite and disjoint sets, called places and transitions, resp.,*
∘ $f \subseteq ((P \times T) \cup (T \times P))$*, the elements of which are called arcs.*
∘ $\lambda : f \rightarrow (Terms \rightarrow \mathbb{N})$*, where* $Terms \rightarrow \{1\}$*. A total function assigning weights to arcs.*
∘ *a marking function* $m_0 : P \rightarrow \mathbb{N}$.

Definition 2. ***(Pre(resp.Post) set places(resp.transitions) of PN)***
Let $PN =< P, T, f, \lambda >$ *be a Petri net,* $p \in P$ *a place then preset and postset of p, noted* $\bullet p$ *and* $p\bullet$*, are defined as follows:*
$\bullet p = \{t \in T | \lambda(t, p) > 0\}$.
$p\bullet = \{t \in T | \lambda(p, t) > 0\}$.

Analogously $\bullet t$ and $t\bullet$ are defined. We also note $\bullet P$ and $P\bullet$ representing pre(resp.post) set of transtions of all the places in set P. Analogously, $\bullet T$ and $T\bullet$ are noted.

Definition 3. ***(Enabled transitions of PN)***

Let m and m' two markings of PN and t a transition in T then $< m, t, m' >$ is a valid firing triplet (represented by $m[t\rangle m'$) iff

1) $\forall p \in \bullet t | m(p) \geq \lambda(p,t)$ (i.e., t is enabled by m).

2) $\forall p \in P, m'(p) = m(p) - \lambda(p,t) + \lambda(t,p)$.

Definition 4. ***(Reading(resp.Non-reading) transitions of PN)***

Let $t \in T$ be a transition in PN. We call t a reading-transition iff its firing does not change the marking of any place $p \in (\bullet t \cup t\bullet)$, i.e., iff $\forall p \in (\bullet t \cup t\bullet), \lambda(p,t) = \lambda(t,p)$. Conversely, we call t a non-reading transition iff $\lambda(p,t) \neq \lambda(t,p)$.

Definition 5. ***(Algebraic Petri net)***

A marked Algebraic Petri Net $APN =< SPEC, P, T, f, \lambda, asg, cond, m_0 >$ consist of

- *an algebraic specification SPEC = (Σ,E),*
- *P and T are finite and disjoint sets, called places and transitions, resp.,*
- *$f \subseteq (P \times T) \cup (T \times P)$, the elements of which are called arcs,*
- *an arc inscription function $\lambda : f \rightarrow (Terms \rightarrow \mathbb{N})$, where $Terms \rightarrow \{T_{OP,asg(p)}\}$,*
- *a sort assignment $asg : P \rightarrow S$,*
- *a function, $cond : T \rightarrow \mathcal{P}_{fin}(\Sigma - equation)$, assigning to each transition a finite set of equational conditions.*
- *an initial marking m_0 assigning a finite multiset over $T_{OP,asg(p)}$ to every place p.*

Definition 6. ***(Enabled transitions of APN)***

Let m and m' two markings of APN and t a transition in T then $< m, t, m' >$ is a valid firing triplet (represented by $m[t\rangle m'$) iff

1) $\forall p \in^\bullet t | m(p) \geq \lambda(p,t)$ (i.e., t is enabled by m).

2)$\forall p \in P, m'(p) = m(p) - \lambda(p,t) + \lambda(t,p)$.

Definition 7. ***(Reading(resp.Non-reading) transitions of APN)***

Let $t \in T$ be a transition in an unfolded APN. We call t a reading-transition iff its firing does not change the marking of any place $p \in ({}^\bullet t \cup t^\bullet)$, i.e., iff $\forall p \in ({}^\bullet t \cup t^\bullet), \lambda(p,t) = \lambda(t,p)$. Conversely, we call t a non-reading transition iff $\lambda(p,t) \neq \lambda(t,p)$.

4 Static Slicing Algorithms

In this section, we will study basic algorithms for static PN slicing [2–5, 7–9]. The objective of every algorithm is to improve the verification process either by reducing a PN model or by partitioning a PN model.

4.1 Chang et al Slicing

Chang et al presented an algorithm for the first time for slicing Petri nets in the context of testing [2]. The presented algorithm slices out all the concurrency set of a Petri net model. The concurrency set is defined as a set of paths in different processes that should be executed concurrently. Based on the information about which parts of the system would be executed, test input data can be generated.

Algorithm 1: Chang Slicing

Input: $ProcessPN[1], ..., ProcessPN[N]$.
$CS = \{t | t \in T, t \; is \; a \; communication \; transition\}$.
Output: $S[1], S[2], \ldots, S[I]$ //A set of concurrency sets.
Variables: $Mar = \{t | t \in T, t \; has \; a \; mark\}$,
$TM = \{t | t \in T, t \; has \; a \; temporary \; mark\}$,
$WS = \{t | t \in CS, t \; is \; in \; current \; process \; being \; scanned\}$.
$Algorithm\_Slicing(ProcessPN[1], \ldots, ProcessPN[N], CS :$
$in; S[1], S[2], \ldots, S[I] : out)$
$forj \leftarrow 1 \; to \; N$ do if there exist more than one path in $ProcessPN[j]$
$then \; changable[j] \leftarrow true$,
$else \; changable[j] \leftarrow false$;
$I \leftarrow 0; \; terminate \leftarrow false$;
$while \; CS \neq \emptyset \; and \; terminate = false \; do$
$I < -I + 1; S[I] \leftarrow \emptyset$;
$Mar \leftarrow \emptyset; TM \leftarrow \emptyset$;
$WS \leftarrow \emptyset; WS \leftarrow WS \cup \{t\}$; /* pick up a t from CS which covers all communication transitions in WS. /*
$Procedure\_findpath(ProcessPN[1], WS : in; PA : out)$; /* input process $ProcessPN[1], WS$, to find a path PA in $ProcessPN[1]$ which covers all communication transitions in WS. If there is no such path, $= PA = \emptyset$ /*
$S[I] \leftarrow S[1] \cup PA$;
$M \leftarrow M \cup \{t | t \in ProcessPN[x], x = 1 \; and \; t \; has \; relation \; with \; PA\}$;
$ProcedureScanning(ProcessPN[1], \ldots, ProcessPN[N] :$
$in; CS, Mar, TM, S[I] : in \; \& \; out)$;
/* scanning all processes according to the base path to find a concurrency set */
$CS \leftarrow CS - CS \cap S[I]$
$end \; while$
$endslicing$

The algorithm first finds a base path that covers at least one communication transition (represented as CS) and adds it into the concurrency set (represented as $S[I]$). To select a path which covers all the marked transitions each is process is scanned.

The path may generate new communication transitions that have relations with the previous process (i.e., been scanned) or the succeeding process that has not been scanned yet. If this path does not involve any new communication transitions CS having relations with the previous processes or these transitions

are already in the concurrency set, then this path is added into the concurrency set and mark those transitions having relations with the succeeding processes. Otherwise, if this path involves new communication transitions having relations with a previous process, say x, to find a new path to cover both marked and temporarily marked transitions. If there is such a path, then replace with the one already in the concurrency set by this new path and mark again the transitions in other process. Otherwise, erase temporary marks and try to find a new path other than the old one that was already in the concurrency set. Afterwards, restart the scanning process from x till all the processes have been scanned and a concurrency set has been found. The procedure is repeated until all the communication transitions are included in the certain concurrency set.

The procedure named $ProcedureScanning(ProcessPN[1], \ldots, ProcessPN[N] : in; CS, Mar, TM, S[I] : in \ \& \ out)$ is central to the slicing construction, by executing this procedure once, a concurrency set can be obtained. We skip the formal description of the procedure and refer the interested reader for the formal description to [2]. Remark that the presented algorithm is a linear time complex and always terminates. The time complexity of the algorithm is $O((n/N))^N$.

4.2 Lee et al Slicing

Lee et al proposed an approach in which Petri nets slices are computed based on the structural concurrency inherent in the Petri nets and compositional reachability graph analysis is performed [5]. The proposed approach may enable verification of properties such as boundedness and liveness, which may fail on unsliced Petri nets due to a state explosion problem.

Algorithm 2: Lee Slicing

$SliceSet = \emptyset$;
$SetofInvariant = find\_minimal\_invariants(PN)$;
$do\{$
$small\_invariant = find\_smallest\_invariant(SetofInvariant)$;
$SliceSet = SliceSet \cup \{small\_invariant\}$;
$SetofInvariant = SetofInvariant - \{small\_invariant\}$
$\}$ $untill$ $(P(SetofInvariant) \subseteq P(SliceSet)$ OR $P(PN) ==$
$P(SliceSet))$;
if $(P(SliceSet) \neq P(PN))\{$
$Uncovered\_P\_Set = P(PN) - P(SliceSet)$;
for $\forall p \in Uncovered\_P\_Set$ do $\{$
$slice = find\_minimally\_connected(SliceSet, p)$;
$slice = slice \cup \{p\}; \}$
$\}$

The basic idea of a slicing algorithm is to slice a Petri net model into a set of Petri net slices using minimal invariants. The algorithm starts with an empty slice set and minimal invariants are selected among $Setofinvariants(by\ find\_minimal\_invariants)$. In the minimal invariant set, the algorithm selects an invariant that has the minimal number of elements ($by\ find\_smallest\_invariant$) and

adds it into the $SliceSet$ until $SliceSet$ covers all the places in PN, $(P(PN) == Place(SliceSet))$ or there exists no minimal invariant which includes a new place $(P(SetofInvariant)P(SliceSet))$.

If the minimal invariant set becomes empty without covering all the places in PN , for each uncovered place, it should be added into a slice to which it is connected by a minimal number of transitions ($byfind\_minimally\_connected$). Like this, the slicing algorithm ensures that every place in PN belongs to some slices.

The proposed algorithm is a linear time complex. The time complexity is $O(N)^3$ and depends on three procedures that are $find\_minimal\_invariant$, $find\_smallest\_invariant$ an $\_connected$. The proposed slicing approach has been applied to dining philosopher and feature interaction problem case studies. By taking the case study of dinning philosopher evaluation of proposed slicing approach is performed. The evaluation criterion is to compare the number of states and transitions between Petri net model, modular Petri nets and sliced Petri nets. The numbers of reachable states and state transitions grow exponentially in Petri nets, the numbers in Modular Petri nets and Petri net slices grow slowly. Remark that there is no huge difference in the growth of states and transitions between the Modular Petri nets and sliced Petri nets.

4.3 Rakow Slicing

A.Rakow, presented two slicing algorithms, which are CTL*$_{-X}$ slicing and safety slicing in [9]. The objective of slicing algorithms is to reduce the size of a net in order to alleviate the state explosion problem for model checking Petri nets. Both algorithms proposed are static and follow backward slicing approach. Before slicing Petri nets, temporal formulas are used to extract the slicing criterion. The slicing criterion consists of concerned places extracted from the temporal formula. A slice is generated by following dependencies backward from the criterion places.

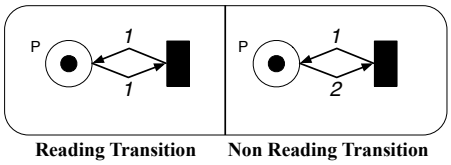

Fig. 4. Reading and non-reading transitions of a PN

In CTL*$_{-X}$ algorithm, Rakow used the concept of *reading* and *non-reading transitions* to generate smaller slice. Informally *reading transitions* are those transitions that do not change the markings of a place while *non-reading transitions* are transitions that change the markings of a place. Excluding the *reading*

transitions and including the *non-reading transitions* during the slicing can certainly reduce the size of a slice.

Algorithm 3: CTL$^*_{-X}$ Slicing

GenerateSlice($\langle P, T, f, \lambda, m_0 \rangle, Crit$){
$T', P_{done} = \emptyset$;
$P' = Crit$;
While $(\exists p \in (P' \setminus P_{done}))$
{ While$(\exists t \in (\bullet p \cup p\bullet) \setminus T') : \lambda(p,t) \neq \lambda(t,p))$ {
$P' = P' \cup \bullet t$;
$T' = T' \cup \{t\}$;
}
$P_{done} = P_{done} \cup \{p\}$;
}
return $\langle P', T', f_{|_{P',T'}}, \lambda_{|_{P',T'}}, m_{0|P'}; \}$

The CTL$^*_{-X}$ algorithm takes a Petri net (PN) and the criterion places ($Crit$) as an input. The algorithm iteratively builds the sliced net by taking all the incoming and outgoing transitions together with their input places. Remark that only the *non-reading transitions* are included in the sliced net. The proposed algorithm is a linear time complex.

Algorithm 4: Safety Slicing

GenerateSlice($\langle P, T, f, \lambda, m_0 \rangle, Crit$){
$T' = \{t \in T \mid \exists p \in Crit : \lambda(p,t) \neq \lambda(t,p)\}$;
$P' = \bullet T \cup Crit$;
$P_{done} = Crit$;
While $(\exists p \in (P' \setminus P_{done}))$
{ While$(\exists t \in (\bullet p \setminus T') : \lambda(p,t) < \lambda(t,p))$ {
$P' = P' \cup \bullet t$;
$T' = T' \cup \{t\}$;
}
$P_{done} = P_{done} \cup \{p\}$;
}
return $\langle P', T', f_{|_{P',T'}}, \lambda_{|_{P',T'}}, m_{0|P'}; \}$

The safety slicing algorithm focuses on the preservation of stutter-invariant linear time safety properties. In contrast to CTL$^*_{-X}$, safety slicing algorithm iteratively take only the transitions that increase the token count on places in the sliced net places and their input places. Remark that the safety slicing does not preserve liveness properties. We took an extermely simple example PN and showed different sliced PNs by applying different slicing algorithms with respect to the criterion place *P3*(see Fig.5).

4.4 Khan et al Slicing

Khan et al presented a slicing algorithm for the first time in the context of algebraic Petri nets (a variant of high-level net) [4]. They argued that the existing

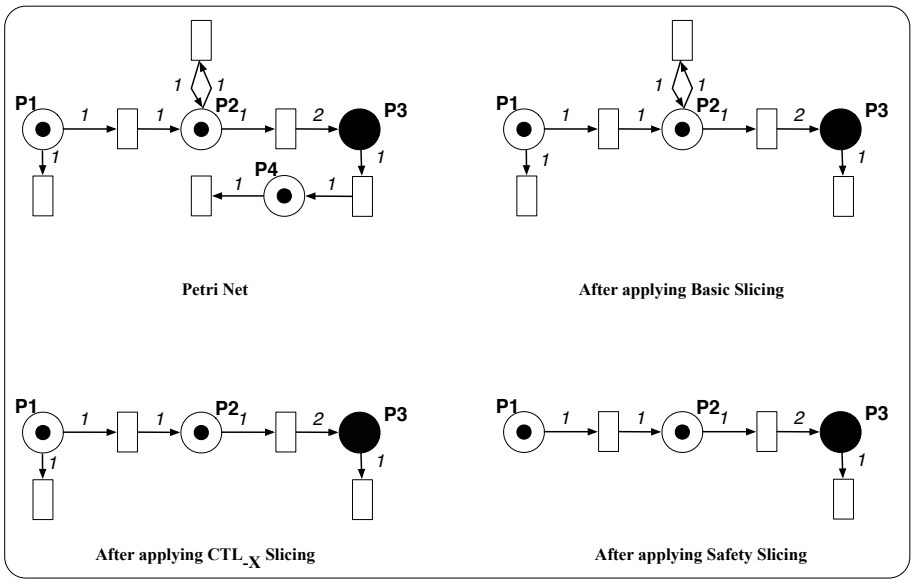

Fig. 5. A PN and sliced PNs by applying different slicing algorithms

slicing constructions are limited to low-level Petri nets and cannot be applied as it is to the high-level Petri nets. In order to be applied to high-level Petri nets they need to be adapted to take into an account the data types.

Algorithm 5: APN Slicing

APNSlicing($\langle SPEC, P, T, f, asg, cond, \lambda, m_0\rangle, Q$){
$T' = \{t \in T \mid \exists p \in Q : t \in (\bullet p \cup p\bullet) : \lambda(p,t) \neq \lambda(t,p)\}$;
$P' = Q \cup \{\bullet T'\}$;
$P_{done} = \emptyset$;
while $((\exists p \in (P' \setminus P_{done}))$ **do**
  while $(\exists t \in (\bullet p \cup p\bullet) \setminus T') : \lambda(p,t) \neq \lambda(t,p))$ **do**
    $P' = P' \cup \bullet t$;
    $T' = T' \cup \{t\}$;
  end
  $P_{done} = P_{done} \cup \{p\}$;
end
return $\langle SPEC, P', T', f_{|_{P',T'}}, asg_{|_{P'}}, cond_{|_{T'}}, \lambda_{|_{P',T'}}, m_{0_{|_{P'}}}\rangle$;
}

In algebraic Petri nets (APNs), the terms may contain variables over the arcs from places to the transitions (resp, transitions to the places) or guard conditions. Authors proposed to unfold the APN to know the ground substitutions of the variables. They used a particular unfolding approach developed by SMV group i.e., a partial unfolding [1]. Perhaps, the proposed approach is independent of any

unfolding approach. The algorithm proposed for slicing APNs starts by taking an unfolded APN and a set of the criterion places.

In APN slicing algorithm, initially T' (representing transitions set of the slice) contains the set of all *pre and post* transitions of the given criterion place. Only the *non-reading* transitions are added to T' set. And P'(representing the places set of the slice) contains all the *preset* places of transitions in T'. The algorithm then iteratively adds other *preset* transitions together with their *preset* places in T' and P'. The proposed algorithm is somewhat similar to the Rakow's algorithm and uses the concept of *reading and non-reading transitions*. By construction it is guaranteed that the algorithm will terminate. We took an extermely simple example APN and showed the unfolding and its sliced APN by applying APNslicing algorithm with respect to the criterion place B(see Fig.6).

To avoid the repeated model checking, a slicing based solution is proposed to reason about the previously satisfied properties [3]. At first, after the evolution of an APN model, slices are built for the evolved and non-evolved APN models with respect to the property by the *APNSlicingEvo* algorithm. The algorithm is somewhat similar to the *APNSlicing* algorithm given above. The main difference is that it takes an APN model as an input instead of the unfolded APN (reducing the overhead of unfolding). And also it does not exclude the *reading transitions* from the slice as syntactically they can not be determined.

Algorithm 6: APN Evo Slicing

APNSlicingEvo($\langle SPEC, P, T, f, asg, cond, \lambda, m_0\rangle, Q$){
$T' = \{t \in T \mid \exists p \in Q : t \in (\bullet p \cup p\bullet)\}$;
$P' = Q \cup \{\bullet T'\}$;
$P_{done} = \emptyset$;
while $((\exists p \in (P' \setminus P_{done}))$ **do**
  while $(\exists t \in (\bullet p \cup p\bullet) \setminus T'))$ **do**
    $P' = P' \cup \bullet t$;
    $T' = T' \cup \{t\}$;
  end
  $P_{done} = P_{done} \cup \{p\}$;
end
return $\langle SPEC, P', T', f_{|_{P',T'}}, asg_{|_{P'}}, cond_{|_{T'}}, \lambda_{|_{P',T'}}, m_{0_{|_{P'}}}\rangle$;
}

Initially, T' (representing transitions set of the slice) contains set of all *pre and post* transitions of the given criterion place. P'(representing places set of the slice) contains all *preset* places of transitions in T'. The algorithm iteratively adds other *preset* transitions together with their *preset* places in T' and P'.

5 Dynamic Slicing Algorithms

In this section, we shall study basic algorithms for dynamic PN slicing presented in the literature [6, 12]. The dynamic slicing algorithms give more reduced Petri

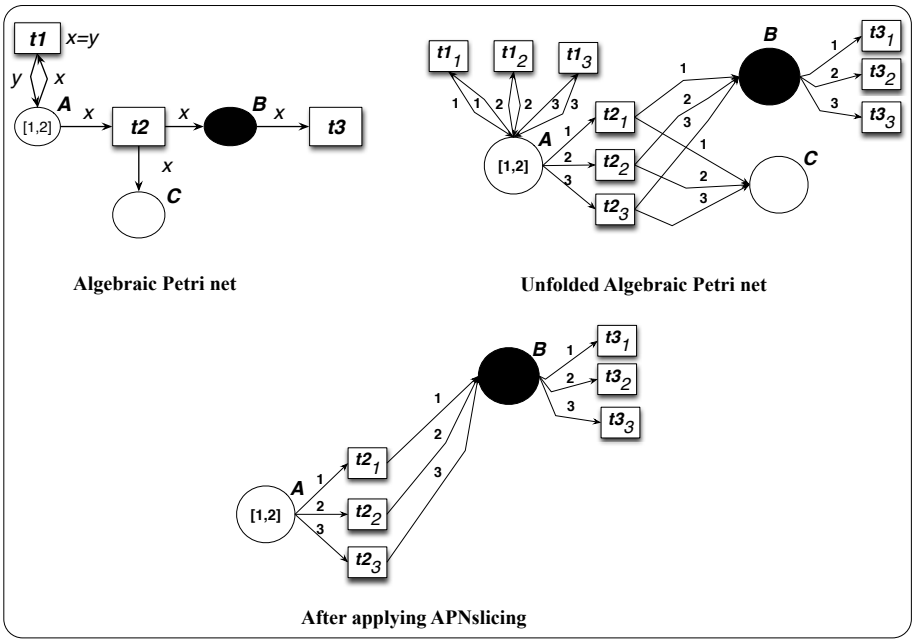

Fig. 6. An APN and its sliced unfolded APN by applying APNslicing algorithm

nets because only the dependences that occur in a specific execution of the Petri net model are taken into account.

5.1 Llorens et al Slicing

Llorens et al presented a dynamic slicing algorithm for the first time [6]. They introduced two different techniques for dynamic slicing of Petri nets. In the first technique, a Petri net and the initial markings are taken into an account while in the second technique firing sequences are fixed to have a more reduced slice. The first technique comprises of three steps. In the first step, the basic algorithm given below computes a backward slice.

Algorithm 7: Bakward Slicing

GenerateSlice($\langle P, T, f, \lambda, m_0\rangle, Crit$){
$T' = \emptyset$;
$P' = Crit$;
While ($\bullet p \neq T'$
{ $T' = T' \cup \bullet P'$;
$P' = P' \cup \bullet T'$;
}
return $\langle P', T', f_{|_{P',T'}}, \lambda_{|_{P',T'}}, m_{0|P'}$; }

Starting from the *criterion* place the algorithm iteratively include all the incoming transitions together with their input places until reaching a fix point. In the second step a forward slicing is computed by the following algorithm.

Algorithm 8: Forward Slicing

GenerateSlice($\langle P, T, f, \lambda, m_0 \rangle$){
$T' = \{t \in T \mid m_0[t\rangle\}$;
$P' = \{p \in P \mid m_0(p) > 0\} \cup T'\bullet$;
$T_{do} = \{t \in T \setminus T' \mid \bullet t \subseteq P'\}$;
While ($T_{do} \neq \emptyset$
{ $T' = T' \cup T_{do}$;
$P' = P' \cup T\bullet_{do}$;
$T_{do} = \{t \in T \setminus T' \mid \bullet t \subseteq P'\}$
}
return $\langle P', T', f_{|_{P',T'}}, \lambda_{|_{P',T'}}, m_{0|P'}; \}$

Starting from the set of initially marked places set the algorithm proceeds further by checking the enabled transitions. Then post set of places are included in the slice. The algorithm computes the paths that may be followed by the tokens of the initial marking.

In the third step both forward and backward slices are intersected to get the resultant slice. By bearing a slight overhead more reduce slices can be obtained. The second technique introduced by Llorens et all fixes the firing sequences and can result smaller slice. The algorithm is defined by auxiliary function and takes initial markings together with the firing sequence denoted by σ and the set of places Q of the slicing criterion.

Algorithm 9: Trace Slicing

$$slice(m_0, \sigma, Q) = \begin{cases} Q & \text{if } i = 0, \\ slice(m_0, \sigma, Q) & \text{if } \forall p \in Q.m_{0-1(p)} \geq m_{0(p)}, i > 0 \\ \{t_i\} \cup slice(m_0, \sigma, Q \cup \bullet t_i) & \text{if} \exists p \in Q.m_{0-1(p)} < m_{0(p)}, i > 0 \end{cases}$$

For a particular marking, a firing sequence and a set of places Q, function slice just moves backwards when no place in Q increased its tokens by the considered firing. Otherwise, the fired transition t_i increased the number of tokens of some place in Q and in this case, function slice already returns this transition t_i and, moreover, it moves backwards also adding the places in $\bullet t_i$ to the previous set Q. When the initial marking is reached, function slice returns the accumulated set of places.

Unfortunately, both techniques are not evaluated to case studies which is a big question mark on the usefulness and practicality.

5.2 Wangyang et al Slicing

Wangyang et al presented a backward dynamic slicing algorithm [12]. The basic idea of proposed algorithm is similar to the algorithm proposed by Lloren

et al [6]. At first for both algorithms, a static backward slice (see algorithm 7) is computed for a given *criterion* place(s). Secondly, in case of Llorens et al a forward slice is computed for the complete Petri net model (see algorithm 8) whereas in case of Wangyang et al forward slice is computed for the resultant Petri net model obtained from static backward slice. Let us suppose that there are n number of places in a Petri net model. After applying the static backward slicing algorithm, let us suppose that there are $n/2$ number of places. The algorithm of Llorens et al compute forward slice for n number of places whereas Wangyang et al algorithm will compute the forward slice only for $n/2$.

Algorithm 10: Wangyang Slicing

Input: *Backward static sliced* PN', m_0.
Output: *Local reachability graph*$(LRG(PN'))$
1. $MP = \{p \in P' \mid m_0(p) > 0\}$, *be the root node, and mark with "New";*
2. *While "New" nodes exist Do*
2.1. *Choose an arbitrary New node as* MP';
2.2. *If* $MP'\bullet = \emptyset$
Then mark MP' *with Terminate node;*
Return to step2;
Endif
2.3. *make that every place* $p \in MP'$ *has a token;*
2.4. *If there does not exist* $t \in T'$ *and is enabled under this situation*
Then mark B' *with Terminate node;*
Return to step2;
Endif
2.5. *Else if there do not exist transition set*
$T_l \subseteq T'$ *and is enabled under this situation*
2.5.1. *For* $t \in T_l$
2.5.1.1. *Compute a new set of places* $MP'' = MP' \setminus \bullet t \cup t\bullet$;
2.5.1.2. *If* MP'' *exists in* $LRG(PN')$
Then create a directed edge from MP' *to* MP''*, mark the edge with t;*
Endif
2.5.1.3. *Else if* MP'' *does not exist in* $LRG(PN')$ *Then create a new node* MP''
and create a directed edge from MP' *to* MP''*, mark edge with t; Endif*
2.5.1.4. *Mark* MP'' *with "New"; Endfor*
Endif
2.6 *Remove mark "New" of* MP'*;*
Repeat

The algorithm starts by taking static backward sliced Petri net model and produce a local reachability graph LRG for the Petri net model. LRG is a directed graph, its node set is the set of places. The mark of an arc is a transition. From the initially marked places a root node is constructed and then enabled transitions are added together with their places. The old node can contribute tokens to new ones then $(LRG(PN'))$ can be obtained by tracking backward static slice forward, and the parts associated with slicing criterion under the ini-

tial marking m_0. Finally, backward dynamic slice can be obtained coupled with the initial marking and corresponding flow relation.

6 Comparison

In this section, the static and dynamic slicing algorithms that were presented earlier are compared and classified. All the PN slicing constructions are proposed for improving the testing or optimizing the model checking of Petri nets. One major difference between the slicing constructions designed for testing and model checking is their *slicing criterion*.

The slicing constructions designed for optimizing model checking extract a *slicing criterion* from the temporal description of the properties. The *slicing criterion* consists of a set of places and then a slice is generated around them. We highlight different slicing algorithms that are designed for improving the model checking with respect to the slice size in the Fig.7. As we can notice that safety slicing algorithm may generate the smallest slice as compared to other algorithms but the scope of safety slicing algorithm is limited to safety properties only. Remark that all the existing slicing algorithms do not allow properties specified with the next time operator. On the other hand the slicing algorithms designed for improving the testing take directly the place or transitions as a *slicing criterion*.

Let us study the results summarized in the table.1, the first column shows different slicing algorithms under observation. For each algorithm, the table lists i) in which context the slicing algorithm is presented i.e., to improve the testing process or to improve the state space of model checking process, ii) reduction affect describing i.e., either the PN model can be reduced or there is no affect of slicing on the model, iii) design context refers to the application of slicing algorithm with respect to Petri nets formalism; there are two major variants of Petri nets that are low-level Petri nets and high-level Petri nets, iv) properties that are preserved by the slicing construction. As some of the algorithms are designed in the context of testing and their objective is to find a particular trace for the analysis, we jointly refer those properties as particular v) slicing type refers to the construction methodology i.e., either it is static or dynamic (see section 2 for slicing types) and is following backward propagation or forward (or both), vi) presenting the time complexity for each construction, vii) whether the algorithm has been implemented or not.

7 Conclusion and Future Work

We have presented a survey of the PN slicing constructions that can be found in the present literature. This work fills the gap for the people who are interested in PN slicing and need more information about the up to date researches. The presented syntactic unification of PN slicing algorithms will facilitate the user to have more clear and easy understanding. By comparing existing PN slicing constructions, we highlighted that most of them are limited to low-level Petri

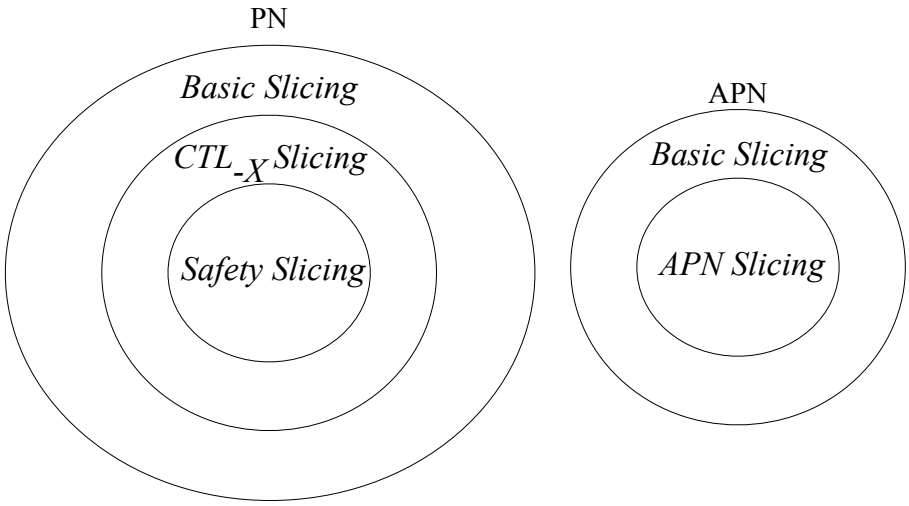

Fig. 7. PN and APN slicing algorithms w.r.t slice size

nets and focusing more on reducing the state space explosion problem for model checking Petri nets. Very few slicing techniques are described in the context of testing. We identified some possible future directions in this domain such as:

- Describing more refined constructions with in the context of testing and more slicing constructions for high-level Petri nets.
- A tool support is very much needed for its practical usability.

References

1. D. Buchs, S. Hostettler, A. Marechal, A. Linard, and M. Risoldi. Alpina: A symbolic model checker. *Springer Berlin Heidelberg*, pages 287–296, 2010.
2. J. Chang and D. J. Richardson. Static and dynamic specification slicing. In *In Proceedings of the Fourth Irvine Software Symposium*, 1994.
3. Y. I. Khan. Optimizing verification of structurally evolving algebraic petri nets. In V. K. A. Gorbenko, A. Romanovsky, editor, *Software Engineering for Resilient Systems*, volume 8166 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013.
4. Y. I. Khan and M. Risoldi. Optimizing algebraic petri net model checking by slicing. *International Workshop on Modeling and Business Environments (ModBE'13, associated with Petri Nets'13)*, 2013.
5. W. J. Lee, H. N. Kim, S. D. Cha, and Y. R. Kwon. A slicing-based approach to enhance petri net reachability analysis. *Journal of Research Practices and Information Technology*, 32:131–143, 2000.
6. M. Llorens, J. Oliver, J. Silva, S. Tamarit, and G. Vidal. Dynamic slicing techniques for petri nets. *Electron. Notes Theor. Comput. Sci.*, 223:153–165, Dec. 2008.

7. A. Rakow. Slicing petri nets with an application to workflow verification. In *Proceedings of the 34th conference on Current trends in theory and practice of computer science*, SOFSEM'08, pages 436–447, Berlin, Heidelberg, 2008. Springer-Verlag.
8. A. Rakow. *Slicing and Reduction Techniques for Model Checking Petri Nets*. PhD thesis, University of Oldenburg, 2011.
9. A. Rakow. Safety slicing petri nets. In S. Haddad and L. Pomello, editors, *Application and Theory of Petri Nets*, volume 7347 of *Lecture Notes in Computer Science*, pages 268–287. Springer Berlin Heidelberg, 2012.
10. W. Reisig. Petri nets and algebraic specifications. *Theor. Comput. Sci.*, 80(1):1–34, 1991.
11. F. Tip. A survey of program slicing techniques. *JOURNAL OF PROGRAMMING LANGUAGES*, 3:121–189, 1995.
12. Y. Wangyang, Y. Chungang, D. Zhijun, and F. Xianwen. Extended and improved slicing technologies for petri nets. *High Technology Letters*, 19(1), 2013.
13. M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.
14. B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, Mar. 2005.

Acknowledgment

This work has been supported by the National Research Fund, Luxembourg, Project MOVERE, ref.C09/IS/02.

8 Appendix

Definition 8. *A signature Σ = (S,OP) consists of a set S of sorts, $OP = (OP_{w,s})_{w\in S^*,s\in S}$ is a $(S^* \times S)-$sorted set of operation names of OP. For ϵ being the empty word, we call $OP_{\epsilon,s}$ the set of constant symbols.*

Definition 9. *A set X of Σ-variables is a family $X = (X_s)_{s\in S}$ of variables set, disjoint to OP.*

Definition 10. *The set of terms $T_{OP,s}(X)$ of sort s is inductively defined by:*

1. $X_s \cup OP_{\epsilon,s} \subseteq T_{OP,s}(X)$;

2. $op(t_1, \ldots, t_n) \in T_{OP,s}(X)$ for $op \in OP_{s_1,\ldots,s_n,s}, n \geq 1$ and $t_i \in T_{OP,s_i}(X)$ (for $i = 1, \ldots, n$).

The set $T_{OP,s} \equiv T_{OP,s}(\emptyset)$ contains the ground terms of sort s, $T_{OP}(X) \equiv \bigcup_{s\in S} T_{OP,s}(X)$ is the set of Σ-terms over X and $T_{OP} \equiv T_{OP}(\emptyset)$ is the set of Σ-ground terms.

Definition 11. *A Σ-equation of sort s over X is a pair (l,r) of terms $l, r \in T_{OP,s}(X)$.*

Definition 12. *An algebraic specification SPEC = (Σ,E) consists of a signature Σ = (S,OP) and a set E of Σ-equations.*

Definition 13. *A Σ-algebra $A = (S_A, OP_A)$ consist of a family $S_A = (A_s)_{s \in S}$ of domains and a family $OP_A = (N_{op})_{op \in OP}$ of operations $N_{op} : A_{s_1} \times \ldots A_{s_n} \rightarrow A_s$ for $op \in OP_{s_1 \ldots s_n, s}$ if $op \in OP_{\epsilon,s}, N_{op}$ congruent to an element of A_s.*

Definition 14. *An assignment of Σ-variables X to a Σ-algebra A is a mapping $ass : X \rightarrow A$,with $ass(x) \in A_s$iff $x \in X_s$. ass is canonically extended to $\overline{ass} : T_{OP}(X) \rightarrow A$, inductively defined by*

1. *$\overline{ass}(x) \equiv ass(x)$ for $x \in X$;*
2. *$\overline{ass}(c) \equiv N_c$ for $c \in OP_{\epsilon,s}$;*
3. *$\overline{ass}(op(t_1, \ldots, t_n)) \equiv N_{op}(\overline{ass}(t_1)), \ldots, \overline{ass}(t_n))$ for $op(t_1, \ldots, t_n) \in T_{OP}(X)$.*

Definition 15. *Let SPEC-algebra is SPEC = (Σ,E) in which all equations in E are valid. Two terms t_1 and t_2 in $T_{OP}(X)$ are equivalent ($t_1 \equiv_E t_2$) iff for all assignments $ass : X \rightarrow A, \overline{ass}(t_1) = \overline{ass}(t_2)$.*

Definition 16. *Let B be a set. A multiset over B is a mapping ms_B: $B \rightarrow \mathbb{N}$. ϵ_B is the empty multiset with $ms_B(x) = 0$ for all $x \in B$. A multiset is finite iff $\{\forall b \in B \mid ms_B(b) \neq 0\}$ is finite.*

Definition 17. *Let $MS_B = \{ms_B$: $B \rightarrow \mathbb{N}\}$ be a set of multisets. The addition function of multisets is denoted by $+ : MS_B \times MS_B \rightarrow MS_B$. Let $ms1_B, ms2_B$ and $ms3_B \in MS_B$. $(ms1_B + ms2_B) = ms3_B \iff \forall b \in B, ms3_B(b) = ms1_B(b) + ms2_B(b)$.*

The subtraction function of multisets is denoted by $- : MS_B \times MS_B \rightarrow MS_B$. Let $ms1_B, ms2_B$ and $ms3_B \in MS_B$. $(ms1_B - ms2_B) = ms3_B \iff \forall b \in B, ms1_B(b) \geq ms2_B(b) \Rightarrow \forall b \in B, ms3_B(b) = ms1_B(b) - ms2_B(b)$.

Definition 18. *Let $MS_B = \{ms_B$: $B \rightarrow \mathbb{N}\}$ be a set of multisets. Let $ms1_B, ms2_B \in MS_B$. We say that $ms1_B$ is smaller than or equal to $ms2_B$ (denoted by $ms1_B \leq ms2_B$) iff*

$\forall b \in B, ms1_B(b) \leq ms2_B(b)$. Further, we say that $ms1_B \neq ms2_B$ iff

$\exists b \in B, ms1_B(b) \neq ms2_B(b)$. Otherwise, $ms1_B = ms2_B$.

Agorithm	*Context*	*Reduction*	*Design context*	*Preserved properties*	*Type slicing*	*Time complexity*	*Implementation*
Chang et al slicing	*Testing*	*No*	*Designed for low-level PN*	*Particular*	*Static backward slicing*	$O((n/N))^N$	*No*
Lee et al slicing	*Model checking*	*No*	*Designed for low-level PN*	*Boundedness and livenss*	*Static backward slicing*	$O(N)^3$	*No*
Rakow CTL$^*_{-X}$ *slicing*	*Model checking*	*Yes*	*Designed for low-level PN*	*CTL*$^*_{-X}$	*Static backward slicing*	$O(2N)$	*Own*
Rakow Safety slicing	*Model checking*	*Yes*	*Designed for low-level PN*	*Safety*	*Static backward slicing*	$O(2N)$	*Own*
Khan et al slicing	*Model checking*	*Yes*	*Designed for high-level PN*	*LTL*$_{-X}$	*Static backward slicing*	$O(2N)$	*Own*
Khan APNevo slicing	*Model checking*	*Yes*	*Designed for high-level PN*	*LTL*$_{-X}$	*Static backward slicing*	$O(2N)$	*No*
Llorens et al APNevo slicing	*Model checking / Testing*	*Yes*	*Designed for low-level PN*	*Particular*	*Dynamic forward/backward slicing*	$O(2T)$	*No*
Llorense et al trace slicing	*Testing*	*Yes*	*Designed for low-level PN*	*Particular*	*Dynamic forward/backward slicing*	$O(2T)$	*No*
Wangyang et al slicing	*Model checking / Testing*	*Yes*	*Designed for low-level PN*	*Particular*	*Dynamic backward slicing*	$O(2T)$	*No*

Table 1. Comparison of PN slicing Algorithms