

Electronic Notes in Theoretical Computer Science 223 (2008) 153–165

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Dynamic Slicing Techniques for Petri Nets[1](#_bookmark0)

# M. Llorens[2](#_bookmark0) J. Oliver[2](#_bookmark0) J. Silva[2](#_bookmark0) S. Tamarit[2](#_bookmark0) G. Vidal[2](#_bookmark0)

*DSIC, Technical University of Valencia Valencia, Spain*

Abstract

Petri nets provide a means for modelling and verifying the behavior of concurrent systems. Program slicing is a well-known technique in imperative programming for extracting those statements of a program that may affect a given program point. In the context of Petri nets, computing a net slice can be seen as a graph reachability problem. In this paper, we propose two slicing techniques for Petri nets that can be useful to reduce the size of the considered net, thereby simplifying subsequent analysis and debugging tasks by standard Petri net techniques.

*Keywords:* Petri nets, program slicing, reachability analysis

# Introduction

*Program slicing* is a method for decomposing programs in order to extract parts of them—called program *slices*—which are of interest. This technique was first defined by Mark Weiser [[20](#_bookmark32)] in the context of program debugging. In particular, Weiser’s proposal was aimed at using program slicing for isolating the program staments that may contain a bug, so that finding this bug becomes simpler for the programmer. In general, slicing extracts the statements that may affect some point of interest, referred to as *slicing criterion*.

Let us illustrate this technique with an example taken from [[19](#_bookmark31)]. Figure [1(a)](#_bookmark1) shows a simple program which requests a positive integer number *n* and computes the sum and the product of the first *n* positive integer numbers. Figure [1(b)](#_bookmark2) shows a slice of this program w.r.t. the slicing criterion (10,product), i.e., variable product in line 10. As can be seen in the figure, all the computations that do not contribute to the final value of the variable product have been removed from the slice.

1 This work has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN under grants TIN2005-09207-C03-02, TIN2008-06622-C03-02, and *Acci´on Integrada* HA2006-0008.

2 Email: [{mllorens,fjoliver,jsilva,stamarit,gvidal}@dsic.upv.es](mailto:%7Bmllorens%2Cfjoliver%2Cjsilva%2Cstamarit%2Cgvidal%7D@dsic.upv.es)

1571-0661 © 2008 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2008.12.037

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 ;**

1. **write (sum) ;**
2. **write (product) ;**
   1. Example program.

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

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

**begin**

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

**end ;**

**write (product) ;**

* 1. Program slice w.r.t. (10,product).

Fig. 1. Sub-figures [1(a)](#_bookmark1) and [1(b)](#_bookmark2) show an example of program slicing.

The work by Weiser has inspired a lot of different approaches to compute slices which include generalizations and concretizations of the initial approach. In general, all of them are classified into two classes: *static* and *dynamic*. A slice is said to be *static* if the input of the program is unknown (this is the case of Weiser’s approach). On the other hand, it is said to be *dynamic* if a particular input for the program is provided, i.e., a particular computation is considered.

In this work, we propose the use of slicing techniques to produce subnets of a Petri net. A Petri net [[13](#_bookmark25),[14](#_bookmark26)] is a graphic, mathematical tool used to model and verify the behavior of systems that are concurrent, asynchronous, distributed, par- allel, non-deterministic and/or stochastic. As a graphic tool, they provide a visual understanding of the system and the mathematical tool facilitates its formal anal- ysis. State space methods are the most popular approach to automatic verification of concurrent systems. In their basic form, these methods explore the transition system associated with the concurrent system. The transition system is a graph, known as the *reachability graph*, that represents the system’s reachable states as nodes: there is an arc from one state *s* to another *s*', whenever the system can evolve from *s* to *s*'. In the worst case, state space methods have to explore all the nodes and transitions in the transition system. This makes the method useless in practice, even though it is simple in concept, due to the state-explosion problem that occurs when a Petri net is applied to nontrivial real problems. The technique is costly even in bounded nets with a finite number of states since, in the worst case, the reachable states are multiplied beyond any primitive recursive function. For this reason, various approaches have been proposed to minimize the number of system states to be studied in a reachability graph [[17](#_bookmark27)].

Program slicing has a great potential here since it allows us to syntactically re- duce a model in such a way that the reduced model is composed only of those parts that may influence the slicing criterion. Since it was originally defined by Weiser, program slicing has been applied to different formalisms which are not strictly pro- gramming languages, like attribute grammars [[18](#_bookmark28)], hierarchical state machines [[9](#_bookmark20)], Z and CSP-OZ specifications [[5](#_bookmark17),[2](#_bookmark13),[3](#_bookmark15)], etc. Unfortunately, very little work has been carried out on slicing for Petri nets (some notable exceptions are [[4](#_bookmark16),[11](#_bookmark23),[15](#_bookmark29),[16](#_bookmark30)]). For

instance, Chang and Wang [[4](#_bookmark16)] present a static slicing algorithm for Petri nets that slices out all sets of paths, known as concurrence sets, so that all paths within the same set should be executed concurrently. In [[11](#_bookmark23)], a static slicing technique for Petri nets is proposed in order to divide enormous P/T nets into manageable modules so that the divided model can be analyzed by a compositional reachability analysis technique. A Petri net model is partitioned into concurrent units (Petri net slices) using minimal invariants. In order to preserve all the information in the original model, uncovered places should be added into minimally-connectable concurrent units since minimal invariants may not cover all the places. Finally, in [[15](#_bookmark29),[16](#_bookmark30)], Rakow presents another static slicing technique to reduce the Petri net size and, thus, lessen the problem of state explosion that occurs in the *model checking*

[[6](#_bookmark18)] of Petri nets [[1](#_bookmark14)]. From the best of our knowledge, there is no previous proposal for *dynamic* slicing of Petri nets. This is surprising because considering an initial

marking and/or a particular sequence of transition firings would allow us to further reduce the size of the slices and focus on a particular use of the considered Petri net.

In this work, we explore two different alternatives for dynamic slicing of Petri nets. Firstly, we present a slicing technique that extends the slicing criterion in [[15](#_bookmark29),[16](#_bookmark30)] in order to also consider an initial marking. We show that this informa- tion can be very useful when analyzing Petri nets and, moreover, it allows us to significantly reduce the size of the computed slice. Furthermore, we show that our algorithm is, in the worst case, as precise as Rakow’s algorithm. This can still be seen as a lightweight approach to slicing since its cost is bounded by the number of transitions in the Petri net. Then, we present a second approach that further reduces the size of the computed slice by only considering a particular execution—here, a sequence of transition firings. Clearly, in this case the computed slice is only use- ful to analyze the considered firing sequence. We illustrate both techniques with examples.

# Petri Nets

A Petri net [[13](#_bookmark25),[14](#_bookmark26)] is a directed bipartite graph, whose two essential elements are called *places* (represented by circles) and *transitions* (represented by bars or rect- angles). The edges of the graph form the *arcs*, which are labelled with a positive integer known as *weight*. Arcs run from places to transitions and vice versa. The *state* of the system modeled by the net is represented by assigning non-negative integers to places. This is known as a *marking*, and is shown graphically by adding small black circles to the places, known as *tokens*. The *dynamic behavior* of the system is simulated by changes in the markings of a Petri net, a process which is carried out by the firing of the transitions. The basic concepts of Petri nets are summarized as follows:

Definition 2.1 A *Petri net* [[13](#_bookmark25),[14](#_bookmark26)] is a tuple N = (*P, T, F* ), where:

* *P* is a set of *places*.
* *T* is a set of *transitions*, such that *P* ∩ *T* = ∅ and *P* ∪ *T* /= ∅.
* *F* is the *flow relation* that assigns weights to arcs: *F* : *P* × *T* ∪ *T* × *P* → N. The *marking M* of a Petri net is defined over the set of places *P* . For each place

*p* ∈ *P* we let *M* (*p*) denote the number of tokens contained in *p*.

A *marked Petri net* Σ is a pair (N *,M* ) where N is a Petri net and *M* is a marking. We denote by *M*0 the *initial marking* of the net.

In the following, given a marking *M* and a set of places *P* , we denote by *M* |*P* the restriction of *M* over *P* , i.e., *M* |*P* (*p*)= *M* (*p*) for all *p* ∈ *P* and *M* |*P* is undefined otherwise.

Definition 2.2 [[14](#_bookmark26)] Given a Petri net N = (*P, T, F* ), we say that a marking *M* '

*covers* a marking *M* if *M* ' ≥ *M* , i.e., *M* '(*p*) ≥ *M* (*p*) for each *p* ∈ *P* .

Given a Petri net N = (*P, T, F* ), we say that a place *p* ∈ *P* is an *input (resp. output) place* of a transition *t* ∈ *T* iff there is an *input (resp. output) arc* from *p* to *t* (resp. from *t* to *p*). Given a transition *t* ∈ *T* , we denote by •*t* and *t*• the set of all input and output places of *t*, respectively. Analogously, given a place *p* ∈ *P* , we denote •*p* and *p*• the set of all input and output transitions of *p*, respectively.

Definition 2.3 Let Σ = (N *,M* ) be a marked Petri net, with N = (*P, T, F* ). We

say that a transition *t* ∈ *T* is *enabled* in *M* , in symbols *M t*

−→

, iff for each input

place *p* ∈ *P* of *t*, we have *M* (*p*) ≥ *F* (*p, t*). A transition may only be fired if it is enabled.

The *ﬁring* of an enabled transition *t* in a marking *M* eliminates *F* (*p, t*) tokens from each input place *p* ∈ •*t* and adds *F* (*t, p*') tokens to each output place *p*' ∈ *t*•,

producing a new marking *M* ', in symbols *M* −→*t*

*M* '.

We say that a marking *Mn* is *reachable* from an initial marking *M*0 if there exists

a *ﬁring sequence σ* = *t t*

*... t*

such that *M*

*t*1 *M*

*t*2 *... t*n

*M* . In this case,

1 2 *n*

0 −→ 1 −→ −→ *n*

we say that *Mn* is reachable from *M*0

*σ*

through *σ*, in symbols *M*0 −→

*Mn*. This

notion includes the empty sequence *ϵ*; we have *M* −→*є*

*M* for any marking *M* . We

say that a firing sequence is *initial* if it starts from an initial marking.

The set of all possible markings which are reachable from an initial marking *M*0 in a marked Petri net Σ = (N *, M*0) is denoted by *R*(N *, M*0) (or simply by *R*(*M*0) when N is clear from the context).

The following notion of *subnet* will be particularly relevant in the context of slicing (roughly speaking, we will identify a slice with a subnet). Let *P* ' × *T* ' ∪ *T* ' × *P* ' ⊆ *P* × *T* ∪ *T* × *P* , we say that a flow relation *F* ' : *P* ' × *T* ' ∪ *T* ' × *P* ' → N is a restriction of another flow relation *F* : *P* ×*T* ∪ *T* ×*P* → N over *P* ' and *T* ', in symbols *F* |(*P* '*,T* '), if *F* ' is defined as follows: *F* '(*x, y*)= *F* (*x, y*) if (*x, y*) ∈ *P* ' × *T* ' ∪ *T* ' × *P* ' and *F* ' is not defined otherwise.

Definition 2.4 [[8](#_bookmark21)] A *subnet* N' = (*P* '*,T* '*,F* ') of a Petri net N = (*P, T, F* ) is a Petri net such that *P* ' ⊆ *P* , *T* ' ⊆ *T* and *F* ' is a restriction of *F* over *P* ' and *T* ', i.e., *F* ' = *F* |(*P* '*,T* ').

# Dynamic Slicing of Petri Nets

In this section, we introduce our first approach to dynamic slicing of Petri nets. We say that our slicing technique is *dynamic* since an initial marking is taken into account (in contrast to previous approaches, e.g., [[4](#_bookmark16),[11](#_bookmark23),[15](#_bookmark29),[16](#_bookmark30)]).

Using an initial marking can be useful, e.g., in debugging. Consider for instance that the user is analyzing a particular trace for a marked Petri net (using a simula- tion tool [[7](#_bookmark19)], which we assume correct), so that an erroneous state is reached. Here, by *erroneous* state, we mean a marking in which some places have an incorrect number of tokens. In this case, we are interested in extracting the set of places and transitions (more formally, a subnet) that may erroneously contribute tokens to the places of interest, so that the user can more easily locate the bug.

Therefore, our first notion of *slicing criterion* is formalized as follows:

Definition 3.1 Let N = (*P, T, F* ) be a Petri net. A *slicing criterion* for N is a pair ⟨*M*0*, Q*⟩ where *M*0 is an initial marking for N and *Q* ⊆ *P* is a set of places.

Roughly speaking, given a slicing criterion ⟨*M*0*, Q*⟩ for a Petri net N, we are interested in extracting a subnet with those places and transitions of N which can contribute to change the marking of *Q* in any execution starting in *M*0.

Our notion of *dynamic* slice is defined as follows. In the following, we say that *σ*' is a *subsequence* of a firing sequence *σ* w.r.t. a set of transitions *T* if *σ*' contains all transitions of *σ* that belong to *T* and in the same order.

Definition 3.2 Let N = (*P, T, F* ) be a Petri net and let ⟨*M*0*, Q*⟩ be a slicing criterion for N. Given a Petri net N' = (*P* '*,T* '*,F* '), we say that N' is a slice of N

w.r.t. ⟨*M*0*, Q*⟩ if the following conditions hold:

* + the Petri net N' is a subnet of N and
  + for each firing sequence *σ* = *t*

*... t*

, for N, with *M*

*t*1 *t*n−1

*... M*

*t*n *M*

1 *n* 0 −→

−→ *n*−1 −→ *n*

such that *Mn*−1(*p*) *< Mn*(*p*) for some *p* ∈ *Q*, there exists a firing sequence *σ*' for

(N '*,M* ' ), with *M* ' = *M*0|*P* ' , such that

0 0

* *σ*' is a subsequence of *σ* w.r.t. *T* ',

' *σ*' '

* *M*0 −→ *Mm*, *m* ≤ *n*, and
* *M* ' covers *Mn*|*P* ' (i.e., *M* '

*m*

*m*

≥ *Mn*|*P* ' ).

Intuitively speaking, a Petri net N' is a slice of another Petri net N if N' is a subnet of N (i.e., no additional places nor transitions are added) and the behaviour of N is preserved in N ' for the restricted sets of places and transitions. In order to formalize this second condition, we require that, for all firing sequences *σ* = *t*1 *... tn* that may *move* tokens to the places of the slicing criterion, i.e.,

*M t*1

*t*n−1 *t*n

0 −→ *...* −→ *Mn*−1 −→ *Mn* and *Mn*−1(*p*) *< Mn*(*p*)*, p* ∈ *Q*

the *restriction* of this firing sequence can also be performed on the slice N ', i.e.,

' *σ*' ' '

*M*0 −→ *Mm* and *Mm* ≥ *Mn*

Trivially, given a Petri net N, the complete net N is always a correct slice w.r.t. any slicing criterion. The challenge then is to produce a slice as small as possible.

Algorithm 1 Dynamic slicing of a marked Petri net.

Let N = (*P, T, F* ) be a Petri net and let ⟨*M*0*, Q*⟩ be a slicing criterion for N. First, we compute a *backward slice* similar to that of [[15](#_bookmark29)]. This is obtained from b sliceN (*Q,* { }), where function b sliceN is defined as follows:

b sliceN (*W, Wdone*)=

⎪⎨ *T* ∪ •*T* ∪ b sliceN (*W* \*W* '

*done*)

if *W* /= { }*,* where *T* = •*p,* and *W* '

*done*

*,W*

*done*

⎧⎪ {} if *W* = {}

'

⎪⎩ for some *p* ∈ *P*

= *Wdone* ∪ {*p*}

Now, we compute a *forward slice* from

f sliceN ({*p* ∈ *P* | *M* (*p*) *>* 0}*,* { }*,* {*t* ∈ *T* | *M t* )

0

where function f sliceN is defined as follows:

⎧ *W* ∪ *R* if *V* = {}

⎪⎪⎨

f sliceN (*W, R, V* )= f sliceN (*W* ∪ *V* •*,R* ∪ *V, V* ')

0 −→}

⎪⎪⎩

if *V* /= { }*,* where *V* ' = {*t* ∈ *T* \(*R* ∪ *V* ) | •*t* ⊆ *W* ∪ *V* •}

Then, the dynamic slice is finally obtained from the intersection of the backward and forward slices. Formally, let

*P* ' ∪ *T* ' = b sliceN (*Q,* { }) ∩ f sliceN ({*p* ∈ *P* | *M* (*p*) *>* 0}*,* { }*,* {*t* ∈ *T* | *M t* )

0

with *P* ' ⊆ *P* and *T* ' ⊆ *T* , the computed slice is

0 −→}

N' = (*P* '*,T* '*,F* |(*P* '*,T* '))

Algorithm [1](#_bookmark4) describes our method to extract a dynamic slice from a Petri net. Intuitively speaking, Algorithm [1](#_bookmark4) constructs the slice of a Petri net (*P, T, F* ) for a set of places *Q* ⊆ *P* as follows. The key idea is to capture a possible token flow relevant for places in *Q*. For this purpose,

* we first compute the possible paths which lead to the slicing criterion,
* then we also compute the paths that may be followed by the tokens of the initial marking.

This can be done by taking into account that (i) the marking of a place *p* depends on its input and output transitions, (ii) a transition may only be fired if it is enabled, and (iii) the enabling of a transition depends on the marking of its input places. The algorithm is divided in three steps:

* The first step is a backward slicing method (which is similar to the *basic slicing algorithm* of [[15](#_bookmark29)]) that obtains a slice N1 = (*P*1*, T*1*, F*1) defined as the subnet of

N that includes all input places of all transitions connected to any place *p* in *P*1, starting with *Q* ⊆ *P*1.

* The core of this method is the auxiliary function b sliceN , which is initially called with the set of places *Q* of the slicing criterion together with an empty set of places.
* For a particular non-empty set of places *W* and a particular place *p* ∈ *W* , function b sliceN returns the transitions *T* in •*p* and the input places of these transitions •*T* . Then, function b sliceN moves backwards adding the place *p* to the set *Wdone* and removing from *W* the updated set *Wdone* until the set *W* becomes empty.
  + The second step is a forward slicing method that obtains a slice N2 = (*P*2*, T*2*, F*2) defined as the subnet of N that includes all transitions initially enabled in *M*0 as well as those transitions connected as output transitions of places in *P*2, starting with *p* ∈ *P* such that *M*0(*p*) *>* 0.
    - We define an auxiliary function f sliceN , which is initially called with the places that are marked at *M*0, an empty set of transitions and the enabled transitions in *M*0.
    - For a particular set of places *W* , a particular set of transitions *R* and a particular non-empty set of transitions *V* , function f sliceN moves forwards adding the places in *V* • to *W* , adding the transitions in *V* to *R* and replacing the set of transitions *V* by a new set *V* ' in which are included the transitions that are not in *R* ∪ *V* and whose input places are in *W* ∪ *V* •.
    - Finally, when *V* is empty, function f sliceN returns the accumulated set of places and transitions *W* ∪ *R*.
  + Finally, the third step obtains the slice N' = (*P* '*,T* '*,F* ') defined as the subnet of N where *P* ' is the intersection of *P*1 and *P*2, *T* ' is the intersection of *T*1 and *T*2, and *F* ' is the restriction of *F* over *P* ' and *T* ', i.e., the intersection of backward and forward slices.

The following result states the completeness of our algorithm for computing Petri net slices. The proof of this result follows easily by induction on the length of the firing sequences considered in Definition [3.2](#_bookmark3).

Theorem 3.3 *Let* N *be a Petri net and* ⟨*M*0*, Q*⟩ *be a slicing criterion for* N *. The dynamic slice* N' *computed in Algorithm* [*1*](#_bookmark4) *is a correct slice according to Deﬁni- tion* [*3.2*](#_bookmark3)*.*

We will now show the usefulness of the technique with a simple example.

Example 3.4 Consider the Petri net N of Fig. [2(a)](#_bookmark6) where the user wants to pro- duce a slice w.r.t. the slicing criterion ⟨*M*0*,* {*p*5*, p*7*, p*8}⟩. Figure [2(b)](#_bookmark7) shows the slice N1 obtained in the first part of Algorithm [1](#_bookmark4). Figure [2(c)](#_bookmark8) shows the slice N2 obtained in the second part of Algorithm [1](#_bookmark4). The subnet shown in Fig. [2(d)](#_bookmark8) is the final result of Algorithm [1](#_bookmark4) (the intersection of N1 and N2). This slice contains all the places and transitions of the original Petri net which can transmit tokens to the slicing criterion.



p10

t9

t10

p0 p1

t5

p3

p8

p2

t0

t1

p4

t7

t4

p9

p6

p5

t2

t6

t8

t3

p7



p10

t9

t10

p0 p1

t5

p3

p8

p2

t0

t1

p4

p6

p5

t2

t3

p7

(a) Initial PN (N *, M*0) (b) Slice (N1*, M*0|P1 )



p10

t9

t10

p0 p1

t5

p3

p8

t0

p4

t7

t4

p9

p6

p5

t2

t6

t8

t3

p7



p10

t9

t10

p0 p1

t5

p3

p8

t0

p4

p6

p5

t2

t3

p7

* 1. Slice (N2*, M*0|P2 )
  2. Slice result of Algorithm [1](#_bookmark4)

Fig. 2. Example of an application of Algorithm [1](#_bookmark4)

Clearly, using an initial marking allows us to produce smaller slices. Surprisingly, previous approaches completely ignored the marking of the net, and thus their slices are often rather big. For instance, the slice of Fig. [2(b)](#_bookmark7) is a subset of the slice produced by Rakow’s algorithm [[15](#_bookmark29)] (this algorithm would also include transitions *t*4, *t*6 and *t*7). Clearly, this slice contains parts of the Petri net that cannot be reached with the given initial marking (e.g., transition *t*1 which could never be fired because place *p*2 is empty). Rakow’s algorithm computes all the parts of the Petri net which could transmit tokens to the slicing criterion and, thus, the associated slicing criterion is just ⟨*Q*⟩, where *Q* ⊆ *P* is a set of places. In contrast, we compute all the parts of the Petri net which could transmit tokens to the slicing criterion from the initial marking. Therefore, our technique is essentially a generalization of Rakow’s technique because the slice produced with Rakow’s algorithm w.r.t. ⟨*Q*⟩ is the same as the slice produced w.r.t. ⟨*M*0*, Q*⟩ if *M*0(*p*) *>* 0 for all *p* ∈ *P* and all *t* ∈ *T* are enabled transitions at *M*0.

Our slicing technique is more general than Rakow’s technique but, at the same time, it keeps its simplicity and efficiency because we still use the Petri net structure to produce the slice. Therefore, our first approach can be considered *lightweight*

because its cost is bounded by the number of transitions *T* of the original Petri net; namely, the cost of our algorithm is O(2*T* ).

# Extracting Slices from Traces

In this section, we present an alternative approach to dynamic slicing that generally produces smaller slices by also considering a particular firing sequence.

In principle, Algorithm [1](#_bookmark4) should consider all possible executions of the Petri net starting from the initial marking. This approach can be useful in some contexts but it is too imprecise for debugging when a particular simulation has been performed. Therefore, in our second approach, we refine the notion of slicing criterion so as to also include the firing sequence that represents the erroneous simulation. By exploting this additional information, the new slicing algorithm will usually produce smaller slices. Formally,

Definition 4.1 Let N = (*P, T, F* ) be a Petri net. A *slicing criterion* for N is a triple ⟨*M*0*, σ, Q*⟩ where *M*0 is a marking for N, *σ* is an initial firing sequence (i.e., starting from *M*0) and *Q* ⊆ *P* is a set of places.

Roughly speaking, given a slicing criterion ⟨*M*0*, σ, Q*⟩ fora Petri net, we are in- terested in extracting a subnet with those places and transitions which are necessary to move tokens to the places in *Q*.

Our notion of *dynamic* slice is defined as follows:

Definition 4.2 Let N = (*P, T, F* ) be a Petri net. Let ⟨*M*0*, σ, Q*⟩ be a slicing criterion for N, with *σ* = *t*1*t*2 *... tn*. Given a Petri net N' = (*P* '*,T* '*,F* '), we say that N ' is a slice of N w.r.t. ⟨*M*0*, σ, Q*⟩ if the following conditions hold:

* + the Petri net N' is a subnet of N,
  + the set of places *Q* appears in *P* ' (i.e., *Q* ⊆ *P* '), and
  + there exists a firing sequence *σ*' for (N'*,M* ' ), with *M* ' = *M*0|*P* ' , such that

0 0

* + - *σ*' is a subsequence of *σ* w.r.t. *T* ',

' *σ*' '

* + - *M*0 −→ *Mm*, *m* ≤ *n*, and
    - *M* ' covers *Mn*|*P* ' (i.e., *M* '

*m*

*m*

≥ *Mn*|*P* ' ).

Trivially, given a marked Petri net (N *, M*0), the complete net N is always a correct slice w.r.t. any slicing criterion. The challenge then is to produce a slice as small as possible.

Intuitively speaking, given a slicing criterion ⟨*M*0*, σ, Q*⟩, the slicing algorithm proceeds as follows:

* + The core of the algorithm lies in the auxiliary function slice, which is initially called with the marking *Mn* which is reachable from *M*0 through *σ*, together with the firing sequence *σ* and the set of places *Q* of the slicing criterion.
  + For a particular marking *Mi*, *i >* 0, a firing sequence *σ* and a set of places *W* , function slice just moves “backwards” when no place in *W* increased its tokens by the considered firing.

Algorithm 2 Extracting slices from traces.

Let N = (*P, T, F* ) be a Petri net and let ⟨*M*0*, σ, Q*⟩ be a slicing criterion for N , with *σ* = *t*1*t*2 *... tn*. Then, we compute a dynamic slice N' of N w.r.t. ⟨*M*0*, σ, Q*⟩ as follows:

* We have N ' = (*P* '*,T* '*,F* '), where *M*0 −→

−→

*t*1

*t*2

*M*1 −→

*... t*n

*Mn*, *P* ' ∪ *T* ' =

slice(*Mn, σ, Q*), *P* ' ⊆ *P* , *T* ' ⊆ *T* , and *F* ' = *F* |(*P* '*,T* '). Auxiliary function slice is defined as follows:

⎧

slice(*Mi, σ,W* ) = ⎪⎨slice(*Mi*−1*, σ,W* ) if ∀*p* ∈ *W. Mi*−1(*p*) ≥ *Mi*(*p*)*, i >* 0

⎪*W* if *i* = 0

⎪⎪⎩{*ti*} ∪ slice(*Mi*−1*, σ,W* ∪ •*ti*) if ∃*p* ∈ *W. Mi*−1(*p*) *< Mi*(*p*)*, i >* 0

* The initial marking *M* ' is the restriction of *M*0 over *P* ', i.e., *M* ' = *M*0|*P* ' .

0 0

* Otherwise, the fired transition *ti* increased the number of tokens of some place in

*W* . In this case, function slice already returns this transition *ti* and, moreover, it moves backwards also adding the places in •*ti* to the previous set *W* .

* Finally, when the initial marking is reached, function slice returns the accumulated set of places (which includes the initial places in *Q*).

We will now show the utility of the technique with a simple example.

Example 4.3 Consider the Petri net N of Example [3.4](#_bookmark5) shown in Fig. [2(a)](#_bookmark6), together with the firing sequence *σ* shown in Fig. [3(b)](#_bookmark12). The firing sequence *σ* = *t*5*t*2*t*3*t*0*t*2*t*3 corresponds to the branch of the reachability graph shown in Fig. [3(a)](#_bookmark11) that goes from the root to the node *M*45. Then, the user can define the slicing criterion

⟨*M*0*, σ,* {*p*5*, p*7*, p*8}⟩ for N; where *M*0 is the initial marking for N defined in Fig [2(a)](#_bookmark6).

Clearly, this slicing criterion focus on a particular execution and thus the slice produced is more precise than the one produced by Algorithm [1](#_bookmark4). In this case, the slice of N w.r.t. ⟨*M*0*, σ,* {*p*5*, p*7*, p*8}⟩ is the Petri net shown in Fig. [3(c)](#_bookmark12).

The following result states the completeness of our algorithm for computing Petri net slices.

Theorem 4.4 *Let* N = (*P, T, F* ) *be a Petri net and let* ⟨*M*0*, σ, Q*⟩ *be a slicing criterion for* N *. The dynamic slice* N ' *computed in Algorithm* [*2*](#_bookmark10) *is a correct slice according to Deﬁnition* [*4.2*](#_bookmark9)*.*

Proof. (Sketch) We prove the claim by induction on the number *n* of transitions in *σ*.

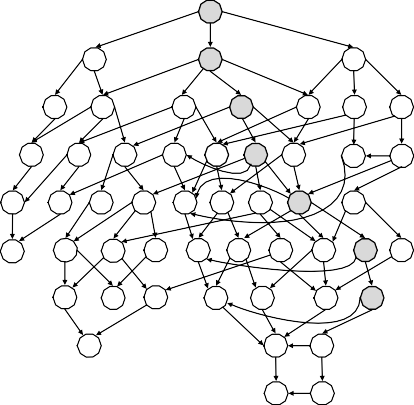
If *n* = 0, then slice(*M*0*, σ, Q*)= *p*∈*Q* slice(*M*0*, σ,* {*p*})= *Q* and the claim follows trivially for N ' = (*Q,* {}*,* {}) and *M* ' = *M*0|*Q*.

0

If *n >* 0, then we distinguish two cases:

* If *Mn*−1(*p*) ≥ *Mn*(*p*) for all *p* ∈ *Q*, then slice(*Mn, σ, Q*) = slice(*Mn*−1*, σ, Q*) and the claim follows by induction.

M0 1 1 0 1 0 1 0 0 1 0 0



t9

M0

t5

t0

M1

t10 t5 t9

M2

t7 t2

M3

t0 t5 t7 t2

M4

M5

t9 M6 t9 M7 t7 M8 t5 M9 t5

t5 t

10

t7

t2

t t

t3 t0

8 0

t

2

M11 M12 M13

M14 M15 t9 M16 M17

t8

M18 t4 M19

t t

7 10

t

8

t

10

t

3

t9

t

t8

0

t4

t2 t

6 t0 t3

t5

t5

M20 M21 M22 M23 M24 M

25

t7

t9

t

6

M

26

t

4

M

27

M28

t8 t10

t3

t4 t2

t

t8

t

t6

t8 t3

t

t0

6

t

t

2

10

t 8

5

7

M29

M30 M31 M32

M33

t3

M34 t9 M35 M36

t7

t

t6 t10 t4 t10 t8

37 38

t5

M41

t

M42

t6 t8

8

M39 M40

t0

M43 M44

t4

t

6

6

4

M1 1 0 0 1 0 1 0 0 1 0 1

M2 1 1 0 0 1 1 0 0 1 0 0

M3 0 0 0 1 1 1 0 0 1 0 0

M17 0 0 0 0 1 0 1 0 1 0 0

M18 0 0 0 1 0 1 0 0 0 0 0

M19 0 0 0 1 0 1 0 1 1 0 0

M20 1 0 0 0 0 1 0 0 1 1 0

M34 0 0 0 0 0 1 0 1 0 1 0

M35 1 1 0 0 0 0 0 0 1 0 0

M36 0 0 0 0 1 0 0 0 1 1 0

M37 0 0 0 0 0 0 1 1 1 0 0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| M10 | M4 | 1 0 0 1 0 1 0 0 2 0 0 | M21 | 1 0 0 0 0 1 0 0 0 0 1 | M38 | 0 0 0 1 0 0 0 0 1 0 0 |
| t3 | M5 | 1 0 0 0 1 1 0 0 1 0 1 | M22 | 1 0 0 0 0 0 1 0 2 0 0 | M39 | 1 0 0 0 0 0 0 0 2 1 0 |

M6 1 1 0 0 0 1 0 0 0 1 0

M23 1 0 0 0 0 1 0 1 1 0 1

M40 1 0 0 0 0 1 0 0 1 0 0

M7 1 1 0 0 0 0 1 0 1 0 0

M8 0 0 0 0 2 1 0 0 1 0 0

M24 M25

0 0 0 0 1 1 0 0 0 0 0

0 0 0 0 0 0 1 0 0 1 0

M41 M42

1 0 0 0 0 0 0 0 1 0 1

0 0 0 0 0 1 0 1 0 0 0

M9 0 0 0 1 0 1 0 0 0 1 0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| M M | M10 | 0 0 0 1 0 0 1 0 1 0 0 | M27 | 0 0 0 0 1 1 0 1 1 0 0 | M44 | 0 0 0 0 1 0 0 0 1 0 0 |
| 4  t3 | M11 | 1 0 0 0 1 1 0 0 2 0 0 | M28 | 0 0 0 1 0 0 0 0 1 1 0 | M45 | 0 0 0 0 0 1 0 2 1 0 0 |
| M45 | M12 | 1 0 0 0 0 1 0 0 0 1 1 | M29 | 1 0 0 0 0 1 0 0 1 0 0 | M46 | 1 0 0 0 0 0 0 0 2 0 0 |

M26 1 1 0 0 0 0 0 0 1 1 0

M43

0 0 0 0 0 0 0 0 0 2 0

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 8  M46 | t10 | t | t8 | M47 | t7  t4 M48 | t | M13  M14 | 1 0 0 0 0 0 1 0 1 0 1  1 1 0 0 0 1 0 0 0 0 0 | M30  M31 | 1 0 0 0 0 1 0 1 2 0 0  1 0 0 0 0 0 0 0 1 1 1 | M47  M48 | 0 0 0 0 0 0 0 0 0 1 0  0 0 0 0 0 0 0 1 1 1 0 |
|  |  |  |  | t8 | t8 |  | M15 | 0 0 0 0 1 1 0 0 0 1 0 | M32 | 1 0 0 0 0 1 0 0 0 0 1 | M49 | 0 0 0 0 0 0 0 0 0 0 0 |
|  |  |  |  | M49 | t M50 |  | M16 | 1 1 0 0 0 1 0 1 1 0 0 | M33 | 0 0 0 0 0 0 1 0 0 0 0 | M50 | 0 0 0 0 0 0 0 1 1 0 0 |

1. Reachability graph

p0 p1

p2 p3

p4

p6



p5

p9 p10

p7 p8

p0 p1 t5 p3 p8



t0

p4

p6

p5

t2

t3

p7

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| M0 | 1 | 1 | 0 | 1 | 0 |  | 1 | 0 | 0 | 1 | 0 | 0 |
|  |  |  |  |  |  |  | t5 |  |  |  |  |  |
|  | 1 | 1 | 0 | 0 | 1 |  | 1 | 0 | 0 | 1 | 0 | 0 |
|  |  |  |  |  |  |  | t2 |  |  |  |  |  |
|  | 1 | 1 | 0 | 0 | 0 |  | 0 | 1 | 0 | 1 | 0 | 0 |
|  |  |  |  |  |  |  | t3 |  |  |  |  |  |
|  | 1 | 1 | 0 | 0 | 0 |  | 1 | 0 | 1 | 1 | 0 | 0 |
|  |  |  |  |  |  |  | t0 |  |  |  |  |  |
|  | 0 | 0 | 0 | 0 | 1 |  | 1 | 0 | 1 | 1 | 0 | 0 |
|  |  |  |  |  |  |  | t2 |  |  |  |  |  |
|  | 0 | 0 | 0 | 0 | 0 |  | 0 | 1 | 1 | 1 | 0 | 0 |
|  |  |  |  |  |  |  | t3 |  |  |  |  |  |
|  | 0 | 0 | 0 | 0 | 0 |  | 1 | 0 | 2 | 1 | 0 | 0 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |





1. Firing sequence *σ* (c) Slice of N w.r.t. ⟨*M*0*, σ,* {*p*5*, p*7*, p*8}⟩

Fig. 3. Example of an application of Algorithm [2](#_bookmark10)

* + Otherwise, there exists some *p* ∈ *Q* with *Mn*−1(*p*) *< Mn*(*p*) and, there- fore, slice(*Mn, σ, Q*) = {*tn*} ∪ slice(*Mn*−1*, σ,Q* ∪ •*tn*). Let N ' = (*P* '*,T* '*,F* '),

*F* ' = *F* |(*P* '*,T* '), and *M* ' = *M*0|*P* ' . Now, we prove that N ' is a slice of N w.r.t.

0

⟨*M*0*, σ, Q*⟩:

* Trivially, N ' is a subnet of N , *M* ' is a restriction of *M*0 and *Q* ⊆ *P* '.

0

* Let N '' be the slice of N w.r.t. ⟨*M ,σ , Q*∪•*t*

⟩, with *σ*

= *M t*1 *M t*2

0 *n*−1 *n*

*... Mn*−1 and N '' = (*P* ''*,T* ''*,F* '').

*n*−1

0 −→ 1 −→

By the inductive hypothesis, there exists a firing sequence *σ*'' for (N ''*,M* ''),

0

with *M* '' = *M*0|*P* '' , such that

0

*σ*'' is a subsequence of *σn*−1 w.r.t. *T* '',

'' *σ*'' ''

*M*0 −→ *Mk* , *k* ≤ *n* − 1, and

*M* '' covers *Mn*−1 (i.e., *M* '' ≥ *Mn*−1).

*k k*

Now, we consider a firing sequence *σ*' for (N'*,M* ' ) that mimicks *σ*'' (which is safe since *P* '' = *P* ' and *T* '' ⊆ *T* ') and then adds one more firing depending on whether *tn* ∈ *T* ' or not. If *σ*' = *σ*'' then the claim follows by induction. Otherwise, it follows trivially by the inductive hypothesis and the fact that *M* ''

0

*k*

covers *Mn*.

# Conclusions and Future Work

In this work, we have introduced two different techniques for dynamic slicing of Petri nets. To the best of our knowledge, this is the first approach to dynamic slicing for Petri nets. The first approach takes into account the Petri net and an initial marking, but produces a slice w.r.t. any possibly firing sequence. The second approach further reduces the computed slice by fixing a particular firing sequence. In general, our slices are smaller than previous (static) approaches where no initial marking nor firing sequence were considered.

As a future work, we plan to carry on an experimental evaluation of our slicing techniques in order to test its viability in practice. We also find it useful to extend our slicing technique to other kind of Petri nets (e.g., coloured Petri nets [[10](#_bookmark22)] and marked-controlled reconfigurable nets [[12](#_bookmark24)]).

# References

1. A. Bell and B.R. Haverkort. Sequential and distributed model checking of Petri nets. *Int. Journal on* *Software Tools for Technology Transfer*, 7(1):43–60, 2005.
2. I. Bru¨ckner. Slicing CSP-OZ Specifications. In P. Pettersson and W. Yi, editors, *Proc. of the 16th* *Nordic Workshop on Programming Theory*, number 2004-041 in Technical Reports of the Department of Information Technology, pages 71–73. Uppsala University, Sweden, 2004.
3. I. Bru¨ckner and H. Wehrheim. Slicing Object-Z Specifications for Verification. In *Proc. of the 4th Int’l Conf. of B and Z Users (ZB 2005)*, pages 414–433. Springer LNCS 3455, 2005.
4. C.K. Chang and H. Wang. A Slicing Algorithm of Concurrency Modeling Based on Petri Nets. In *Proc. of the Int’l Conf. on Parallel Processing (ICPP’86)*, pages 789–792. IEEE Computer Society Press, 1986.
5. J. Chang and D. Richardson. Static and dynamic specification slicing. In *Proc. of the Fourth Irvine Software Symposium*. Irvine, CA, 1994.
6. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press: Cambridge, MA, 2000.
7. Petri Nets Tool Database. Available at

<http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html>.

1. J. Desel and J. Esparza. *Free choice Petri nets*. Cambridge University Press, New York, NY, USA, 1995.
2. M.P.E. Heimdahl and M.W. Whalen. Reduction and Slicing of Hierarchical State Machines. In

M. Jazayeri and H. Schauer, editors, *Proc. of the 6th European Software Engineering Conference (ESEC/FSE’97)*, pages 450–467. Springer LNCS 1301, 1997.

1. K. Jensen. Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1: Basic Concepts, 1992. Volume 2: Analysis Methods, 1994. Volume 3: Practical Use, 1997. Monographs in Theoretical Computer Science, Springer-Verlag.
2. W.J. Lee, S.D. Cha, Y.R. Kwon, and H.N. Kim. A Slicing-based Approach to Enhance Petri Net Reachability Analysis. *Journal of Research and Practice in Information Technology*, 32(2):131–143, 2000.
3. M. Llorens and J. Oliver. Introducing Structural Dynamic Changes in Petri Nets: Marked-Controlled Reconfigurable Nets. In Farn Wang, editor, *Proc. of the 2nd Int’l Conf. on Automated Technology for Verification and Analysis (ATVA’04)*, pages 310–323. Springer LNCS 3299, 2004.
4. T. Murata. Petri Nets: Properties, Analysis and Applications. *Proc. of the IEEE*, 77(4):541–580, 1989.
5. J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.
6. A. Rakow. Slicing Petri Nets. Technical report, Department fu¨r Informatik, Carl von Ossietzky Universit¨at, Oldenburg, 2007.
7. A. Rakow. Slicing Petri Nets with an Application to Workflow Verification. In *Proc. of the 34th Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM 2008)*, pages 436–447. Springer LNCS 4910, 2008.
8. M. Rauhamaa. *A Comparative Study of Methods for Efficient Reachability Analysis*. Licentiate’s thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Digital Systems Laboratory, 1990.
9. A.M. Sloane and J. Holdsworth. Beyond traditional program slicing. In *Proc. of the Int’l Symp. on Software Testing and Analysis*, pages 180–186, San Diego, CA, 1996. ACM Press.
10. F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.
11. M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.