

Slicing High-level Petri nets

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. High-level Petri nets (evolutions of low-level Petri nets) are well suitable formalisms to represent complex data, which influence the behavior of distributed, concurrent systems. However, usual verification techniques such as model checking and testing remain an open challenge for both (i.e., low-level and high-level Petri nets) because of the state space explosion problem and test case selection. The contribution of this paper is to propose a technique to improve the model checking and testing of systems modeled using Algebraic Petri nets (a variant of high-level petri nets). To achieve the objective, we propose different slicing algorithms for Algebraic Petri nets. We argue that our slicing algorithms significantly improve the state of the art related to slicing APNs and can also be applied to low-level Petri nets with slight modifications. We exemplify our proposed algorithms through a case study of a car crash management system.

Key words: High-level Petri nets, Model checking, Testing, Slicing

1 Introduction

Petri nets are well known low-level formalism for modeling and verifying distributed, concurrent systems. The major drawback of low-level Petri nets formalism is their inability to represent complex data, which influences the behavior of a system. Various evolutions of low-level Petri nets (PNs) have been created to raise the level of abstraction of PNs. Among others, high-level Petri nets (HLPNs) raise the level of abstraction of PNs by using complex structured data [17]. However, HLPN can be unfolded into a behaviourally equivalent PNs.

For the analysis of concurrent and distributed systems (including which are modeled using PNs or HLPNs) model checking is a common approach, consisting in verifying a property against all possible states of a system. However, model checking remains an open challenge for both (PNs & HLPNs) because of 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 both in terms of time and memory consumption [2, 4, 11, 20]. This is particularly true for HLPN models, as the use of complex data (with possibly large associated data domains) makes the number of states grow very quickly.

An intense field of research is targeting to find ways to optimize model checking, either by reducing the state space or by improving the performance of model checkers. In recent years major advances have been made by either modularizing the system or by reducing the states to consider (e.g., partial orders, symmetries). The symbolic model checking partially overcomes this problem by encoding the state space in a condensed way by using *Decision Diagrams* and has been successfully applied to PNs [1, 2]. Among others, Petri net slicing (*PN slicing*) has been successfully used to optimize model checking and testing [3, 7, 10, 12–16, 21]. *PN slicing* is a syntactic technique used to reduce a Petri net model based on the given *criteria*. The given *criteria* refer to the point of interest for which the Petri net model is analyzed. The sliced part constitutes only that part of the Petri net model that may affect the analysis based on the criteria..

One limitation of the proposed slicing algorithms that are designed to improve the model checking in the literature so far is that most of them are only applicable to low-level Petri nets. Recently, an algorithm for slicing APNs has been proposed [10]. We extend their proposal and introduced a new slicing algorithm. By evaluating and comparing both algorithms, we showed that our slicing algorithm significantly improves the model checking of APNs. Another limitation of the proposed slicing algorithms that are designed to improve the testing is that they are limited to low-level Petri nets. We define a slicing algorithm for the first time in the context of testing for APNs. The objective is to reduce the effort of generating large test input data by generating a smaller net. We highlight the significant differences of different slicing constructions (designed for improving model checking or testing) and their evaluations and applications contexts. Our slicing algorithms can also be applied to low-level Petri nets with some slight modifications.

The remaining part of the paper is structured as follows: in section 2, we give formal definitions necessary for the understanding of proposed slicing algorithms. In section 3, different slicing algorithms are presented together with their informal and formal descriptions. In section 4, we discuss related work and we give a comparison with existing approaches. A small case study from the domain of crisis management system (a car crash management system) is taken to exemplify the proposed slicing algorithms in section 5. An experimental evaluation of the proposed algorithms is performed in section 6. In section 7, we draw conclusions and discuss future work.

2 Basic Definitions

Algebraic Petri nets are an evolution of low-level Petri nets. APNs have two aspects, i.e., 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) [5, 15, 17, 18] (Note: we refer the interested reader to [9] for the details on algebraic specifications used in the formal definition of APNs for our work.) .

Definition 1. A marked Algebraic Petri Net $APN = \langle SPEC, P, T, f, asg, cond, \lambda, m_0 \rangle$ consist of

- an algebraic specification $SPEC = (\Sigma, E)$, where signature Σ consists of sorts S and operation symbols OP and E is a set of Σ equations defining the meaning of operations,
- 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,
- a sort assignment $asg : P \rightarrow S$,
- a function, $cond : T \rightarrow \mathcal{P}_{fin}(\Sigma - \text{equation})$, assigning to each transition a finite set of equational conditions.
- an arc inscription function λ assigning to every (p, t) or (t, p) in f a finite multiset over $T_{OP, asg(p)}$, where $T_{OP, asg(p)}$ are algebraic terms (if used “closed”(resp. free) terms to indicate if they are build with sorted variables closed or not),
- an initial marking m_0 assigning a finite multiset over $T_{OP, asg(p)}$ to every place p .

Definition 2. The preset of $p \in P$ is $\bullet p = \{t \in T \mid (t, p) \in f\}$ and the postset of p is $p^\bullet = \{t \in T \mid (p, t) \in f\}$. The pre and post sets of $t \in T$ defined as: $\bullet t = \{p \in P \mid (p, t) \in f\}$ and $t^\bullet = \{p \in P \mid (t, p) \in f\}$.

Definition 3. Let m and m' two markings of APN and t a transition in T then $\langle m, t, m' \rangle$ is a valid firing triplet (denoted by $m[t]m'$) iff

- 1) $\forall p \in \bullet t \mid m(p) \geq \lambda(p, t)$ (i.e., t is enabled by m).
- 2) $\forall p \in P \mid m'(p) = m(p) - \lambda(p, t) + \lambda(t, p)$.

3 Slicing Algorithms

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. Considering a property over a Petri net, we are interested to define a syntactically smaller net that could be equivalent with respect to the satisfaction of the property of interest. To do so the slicing technique starts by identifying the places directly concerned by the property. Those places constitute the *slicing criterion*. The algorithm then keeps 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 iteratively repeated for the latter places, until reaching a fixed point. Roughly, we can divide PN slicing algorithms into two major classes, which are *Static Slicing algorithms* and *Dynamic Slicing algorithms*. An algorithm is said to be static if the initial markings of places are not considered for building the slice. Only a set of places is considered as a *slicing criterion*. The *Static Slicing algorithms* starts from the given criterion place and includes all the pre and post set of transitions together with their incoming places. There may exist sequence of transitions in the sliced net that are not fireable because their incoming places are initially empty and do not get markings from any other way. An algorithm is said to be *dynamic slicing algorithm*, if the initial markings of places are considered for building the slice. The *slicing criterion* will utilize the available information of initial markings and produce a smaller sliced net. For a given *slicing criterion* that consists of

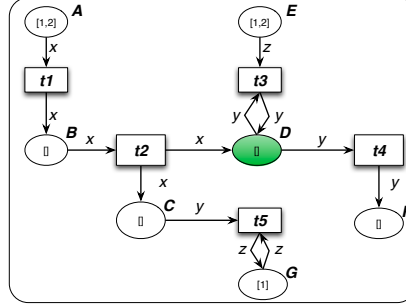


Fig. 1. An example APN model (*APNexample*)

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 criterion place in any execution starting from the initial marking. The sliced net will exclude sequence of transitions in the resultant slice that are not fireable because their incoming places are not initially marked and do not get markings from any other way.

One characteristic of APNs that makes them complex to slice is the use of multiset of algebraic terms over the arcs. In principle, algebraic terms may contain the variables. Even though, we want to reach a syntactically reduced net, its reduction by slicing, needs to determine the possible ground substitutions of these algebraic terms.

We follow [10] to partially unfold the APN first and then perform the slicing on the unfolded APN. In general, unfolding generates all possible firing sequences from the initial marking of the APN. The AIPiNA tool (a symbolic model checker for Algebraic Petri nets) allows user to define partial algebraic unfolding and presumed bounds for the infinite domains [1], using some aggressive strategies for reducing the size of large data domains. The complete description of the partial unfolding for APNs is out of the scope, for further details and description about the partial unfolding used in our approach, we refer the interested reader to follow [1, 10]. The Fig. 1 shows an APN model, all the places and variables over the arcs are of sort *naturals* (defined in the algebraic specification of the model, and representing the \mathbb{N} set). Since the \mathbb{N} domain is infinite (or anyway extremely large even in its finite computer implementations), it is clear that it is impractical to unfold this net by considering all possible bindings of the variables to all possible values in \mathbb{N} . However, given the initial marking of an APN and its structure it is easy to see that none of the terms on the arcs (and none of the tokens in the places) will ever assume any natural value above 3. For this reason, following [1], we can set a *presumed bound* of 3 for the *naturals* data type, greatly reducing the size of the data domain. By assuming this bound, the unfolding technique in [1] proceeds in three steps. First, the data domains of the variables are unfolded up to the presumed bound. Second, variable bindings are computed, and only those are kept that satisfy the guards. Third, the computed

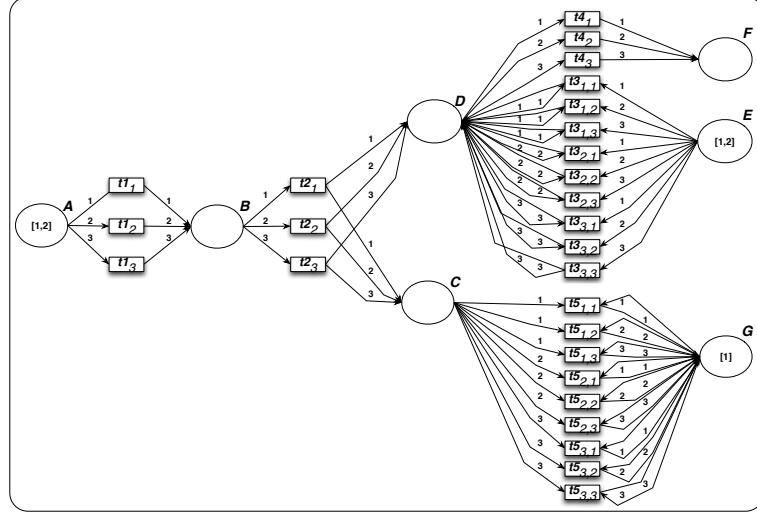


Fig. 2. The unfolded example APN model (*UnfoldedAPN*)

bindings are used to instantiate a binding-specific version of the transition. The resulting unfolded APN model of Fig.1 is shown in the Fig. 2. The transitions arcs are indexed with the incoming and outgoing values of tokens.

3.1 Abstract Slicing on Unfolded APNs

Abstract slicing has been defined as a *static slicing algorithm*. The objective is to improve the model checking of APNs. In the previous static algorithm proposed for APNs, the notions of *reading* and *non-reading transitions* are applied to generate a smaller sliced net. The basic idea of *reading* and *no-reading transitions* was coined by Astrid Rakow in the context of PNs [16], and later adapted in the context of APNs in [10]. Informally, the *reading transitions* are transitions that are not subject to change the marking of a place. On the other hand the *non-reading transitions* change the markings of a place (see Fig.3). To identify a transition to be a reading or non-reading in a low-level or high-level Petri nets, we compare the arcs inscriptions attached over the incoming and outgoing arcs. Excluding *reading transitions* and including only *non-reading transitions* reduces the slice size.

Definition 4. (*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 $\lambda(p, t) \neq \lambda(t, p)$.

We extend the existing slicing operators by introducing the notion of *neutral transitions* and using them with the *reading transitions*. Informally, a *neutral*

transition consumes and produces the same token from its incoming place to an outgoing place. The cardinality of incoming (resp.) outgoing arcs of a neutral tranistion is strictly equal to one and the cardinality of outgoing arcs from an incoming place of a neutral transition is equal to one as well.

Definition 5. (Neutral transitions of APN) Let $t \in T$ be a transition in an unfolded APN. We call t a neutral-transition iff it consumes token from a place $p \in {}^\bullet t$ and produce the same token to $p' \in t^\bullet$, i.e., $t \in T \wedge \exists p \exists p' / p \in {}^\bullet t \wedge p' \in t^\bullet \wedge |p^\bullet| = 1 \wedge |{}^\bullet t| = 1 \wedge |t^\bullet| = 1 \wedge \lambda(t, p) = \lambda(t, p')$.

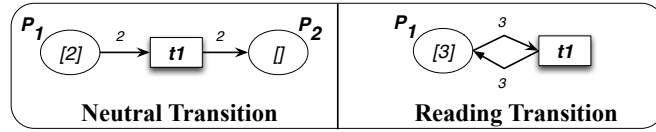


Fig. 3. Neutral and Reading transitions of Unfolded APN

Abstract Slicing Algorithm: The abstract slicing algorithm starts with an unfolded APN and a slicing criterion $Q \subseteq P$ containing criterion place(s). We build a slice for an unfolded APN based on Q by applying the following algorithm:

Algorithm 1: Abstract slicing algorithm

```

AbsSlicing( $\langle SPEC, P, T, F, asg, cond, \lambda, m_0 \rangle, Q$ ) {
   $T' \leftarrow \{t \in T / \exists p \in Q \wedge t \in ({}^\bullet p \cup p^\bullet) \wedge \lambda(p, t) \neq \lambda(t, p)\}$ ;
   $P' \leftarrow Q \cup \{{}^\bullet T'\}$  ;
   $P_{done} \leftarrow \emptyset$  ;
  while  $((\exists p \in (P' \setminus P_{done}))$  do
    while  $(\exists t \in (({}^\bullet p \cup p^\bullet) \setminus T') \wedge \lambda(p, t) \neq \lambda(t, p))$  do
       $P' \leftarrow P' \cup \{{}^\bullet t\}$ ;
       $T' \leftarrow T' \cup \{t\}$ ;
    end
     $P_{done} \leftarrow P_{done} \cup \{p\}$ ;
  end
  while  $(\exists t \exists p \exists p' / t \in T' \wedge p \in {}^\bullet t \wedge p' \in t^\bullet \wedge |{}^\bullet t| = 1 \wedge |t^\bullet| = 1 \wedge |p^\bullet| = 1$ 
   $\wedge p \notin Q \wedge p' \notin Q \wedge \lambda(p, t) = \lambda(t, p'))$  do
     $m(p') \leftarrow m(p') \cup m(p)$ ;
    while  $(\exists t' \in {}^\bullet p / t' \in T')$  do
       $\lambda(p'^\bullet, p) \leftarrow \lambda(p'^\bullet, p') \cup \lambda(t', p)$ ;
    end
     $T' \leftarrow T' \setminus \{t \in T' / t \in p^\bullet \wedge t \in {}^\bullet p'\}$ ;
     $P' \leftarrow P' \setminus \{p\}$ ;
  end
  return  $\langle SPEC, P', T', F|_{P', T'}, asg|_{P'}, cond|_{T'}, \lambda|_{P', T'}, m_0|_{P'} \rangle$ ;
}
```

In the Abstract slicing algorithm, initially T' (representing transitions set of the slice) contains a set of all the *pre and post* transitions of the given criterion places. Only the *non-reading transitions* are added to T' . P' (representing the places set of the slice) contains all the *preset* places of the transitions in T' . The algorithm then iteratively adds other *preset* transitions together with their *preset* places in the T' and P' . Then the *neutral transitions* are identified and their *pre and post* places are merged to one place together with their markings.

Considering the APN-Model shown in fig. 1, let us now apply our proposed algorithm on two example properties (i.e., one from the class of *safety* properties and one from *liveness* properties). Informally, we can define the properties:

φ_1 : “The values of tokens inside place D are always smaller than 5”.

φ_2 : “Eventually place D is not empty”.

Formally, we can specify both properties in the *CTL* as:

$\varphi_1 = \mathbf{AG}(\forall token \in m(D)/token < 5)$.

$\varphi_2 = \mathbf{AF}(|m(D)| \neq \emptyset)$.

For both properties, the slicing criterion $Q = \{D\}$, as D is the only place concerned by the properties. The resultant sliced net can be observed in fig.4, which is smaller than the original unfolded net (shown in fig.2).

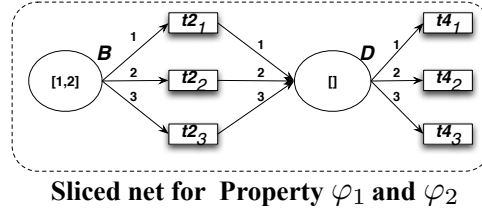


Fig. 4. The sliced unfolded APNs (by applying *abstract slicing*)

Table 1. Comparison of number of states required to verify the property with and without abstract slicing

Properties	No of states required without slicing	No of states required with slicing
φ_1	148	9
φ_2	148	9

Let us compare the number of states required to verify the given property without slicing and after applying abstract slicing. In the first column of Table.1, number of states are given that are required to verify the property without slicing and in the second column number of states are given to verify the property by slicing.

The *abstract slicing* can be applied to low-level Petri nets with slight modifications. The *criteria* to build *abstract slice* for both formalisms (i.e., Algebraic Petri nets and low-level Petri nets) remain the same. In case of low-level Petri nets, we do not unfold the net and the slice is built directly. The idea of including *non-reading transitions* together with merging of places by identifying *neutral transitions* remains the same for both formalisms. (Note: we refer the interested reader to [9] for the proof of preservation of properties by applying the *Abstract slicing algorithm*.)

Abstract Slicing on APN without unfolding : *Abstract slicing* extends the previous proposal of APNs slicing by unfolding the APN and then slicing the unfolded APN. One major criticism on *abstract slicing* and previous slicing construction is the complexity of unfolding APNs. As discussed in the previous section, APNs are unfolded to identify the *reading transitions*(resp. *neutral transitions*) such that a smaller sliced net can be obtained. We can avoid the complexity of unfolding APNs and can perform slicing directly on APNs with a slight trade-off. It is important to note that by applying *abstract slicing* directly on APNs, the sliced net may end up with some *reading transitions* (resp. *neutral transitions*) included. This is due to the fact that the arc inscriptions are syntactically compared to identify *reading transitions*(resp. *neutral transitions*) in slicing algorithm. In Fig.5, two *reading transitions*(resp. *neutral transitions*) can be observed, *abstract slicing* will not consider the transition (shown in the right side of the figure 5) as a *reading transition*(resp. *neutral transitions*). This is a slight trade off to avoid the complexity of unfolding. It is a rare situation to have syntactically *non-reading transitions*(resp. *non-neutral transitions*) which are semantically *reading transitions*(resp. *neutral transitions*). The *Abstract slicing algorithm* can be directly applied to APNs without any change in the syntax.

3.2 Concerned Slicing

Concerned slicing algorithm has been defined as a *dynamic slicing algorithm*. The objective is to extract a subnet with those places and transitions of the APN model that can contribute to change the markings of a given *criterion* place in any execution starting from the initial markings. Concerned slicing can be useful in debugging. Consider for instance that the user is analyzing a particular trace of the marked APN model (using a simulation tool) so that erroneous state is reached.

The *slicing criterion* to build the concerned slice is different as compared to the *abstract slicing* algorithm. In the *concerned slicing* algorithm, available

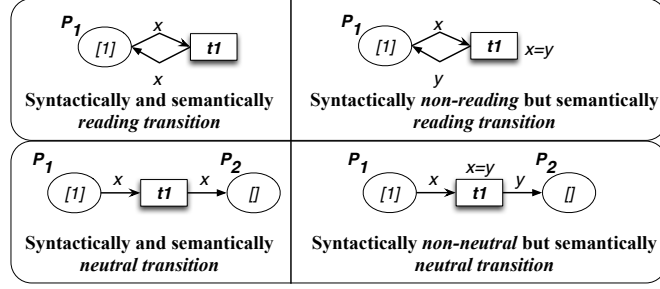


Fig. 5. Syntactically reading (resp. neutral) and non-reading (resp. non-neutral) transitions of APNs

information about the initial markings is utilized and it is directly applied to APNs instead of their unfoldings.

Algorithm 2: Concerned slicing algorithm

```

ConcernedSlicing( $\langle SPEC, P, T, F, asg, cond, \lambda, m_0 \rangle, Q$ ) {
   $T' \leftarrow \emptyset$ ;
   $P' \leftarrow Q$ ;
  while ( $\bullet P \neq T'$ ) do
     $P' \leftarrow P' \cup \bullet T'$ ;
     $T' \leftarrow T' \cup \bullet P'$ ;
  end
   $T'' \leftarrow \{t \in T' / m_0[t] > 0\}$ ;
   $P'' \leftarrow \{p \in P' / m_0(p) > 0\} \cup T''^\bullet$ ;
   $T_{do} \leftarrow \{t \in T' \setminus T'' / \bullet t \subseteq P''\}$ ;
  while ( $T_{do} \neq \emptyset$ ) do
     $P'' \leftarrow P'' \cup T_{do}^\bullet$ ;
     $T'' \leftarrow T'' \cup T_{do}$ ;
     $T_{do} \leftarrow \{t \in T' \setminus T'' / \bullet t \subseteq P''\}$ ;
  end
  return  $\langle SPEC, P'', T'', F|_{P'', T''}, asg|_{P''}, cond|_{T''}, \lambda|_{P'', T''}, m_0|_{P''} \rangle$ ;
}
```

Starting from the *criterion* place the algorithm iteratively include all the incoming transitions together with their input places until reaching a fix point. Then starting from the set of initially marked places set the algorithm proceeds further by checking the enabled transitions. Then the 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.

Considering the APN-Model shown in fig. 1, let us now take the place D as criterion and apply our proposed algorithm on it. The resultant sliced APN-Model is shown in the fig. 6. The test input data can be generated for the sliced APN-model to observe which tokens are coming to the criterion place.

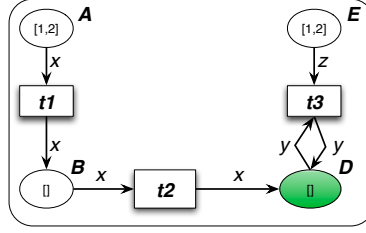


Fig. 6. The sliced APN by applying *concerned slicing*

4 Related Work

The term slicing was coined by M.Weiser for the first time in the context of program debugging [22]. 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 a program.

To explain the basic idea of *program slicing* according to Wieser [22], let us consider an example program shown in the Fig.7. The Fig.7(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, $C = (line10, \{product\})$.

The Fig.7(b) shows the sliced program that is obtained by tracing backwards possible influences on the variables: In the line 7, *product* is multiplied by *i*, and in the 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 (The interested reader can find more details about the *program slicing* from [19,23]).

(1) read(n) ;	read(n) ;
(2) i := 1 ;	i := 1 ;
(3) sum := 0 ;	
(4) product := 1 ;	product := 1 ;
(5) while i <= n do	while i <= n do
begin	begin
(6) sum := sum + i ;	
(7) product := product * i ;	product := product * i ;
(8) i := i + 1 ;	i := i + 1 ;
end ;	end ;
(9) write (sum) ;	
(10) write (product) ;	write (product) ;
(a) Example program.	(b) Program slice w.r.t. (10,product).

Fig. 7. An example program and sliced program w.r.t. given *criterion*

The first algorithm about Petri net slicing was presented by Chang et al [3]. They proposed an algorithm on Petri nets testing that slices out all sets of paths, called concurrency sets, such that all paths within the same set should be executed concurrently. Lee et al. proposed a Petri nets slicing approach to partition huge place/transition net models into manageable modules such that the partitioned model can be analyzed by compositional reachability analysis technique [12]. Llorens et al. introduced two different techniques for dynamic slicing of Petri nets [13]. In the first technique, the Petri net and an initial marking is taken into account, 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. Wangyang et al presented a backward dynamic slicing algorithm [21]. The basic idea of the proposed algorithm is similar to the algorithm proposed by Llorens et al, [13]. At first for both algorithms, a static backward slice is computed for a given *criterion* place(s). Secondly, in the case of Llorens et al a forward slice is computed for the complete Petri net model whereas in the case of Wangyang et al, a forward slice is computed for the resultant Petri net model obtained from the static backward slice.

Astrid Rakow developed two algorithms for slicing Petri nets i.e., CTL^*_X slicing and *Safety slicing* in [16]. The key idea behind the construction is to distinguish between *reading* and *non-reading* transitions. A reading transition $t \in T$ can not change the token count of a place $p \in P$ while other transitions are called *non-reading transitions* as they change the token account. For the CTL^*_X slicing, a subnet is built iteratively by taking all non-reading transitions of a place P together with their input places, starting with the given criterion place. For the *Safety slicing* a subnet is built by taking only transitions that increase token count on the places in P and their input places. The CTL^*_X slicing algorithm is fairly conservative. By assuming a very weak fairness assumption on Petri net it approximates the temporal behavior quite accurately by preserving all the CTL^*_X properties and for the safety slicing focus is on the preservation of stutter-invariant linear safety properties only.

Khan et al presented a slicing technique for algebraic Petri nets [10]. They argued that all the 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 account the data types. In algebraic Petri nets (APNs), terms may contain the variables over the arcs from place to transitions (or transitions to 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 the 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 the criterion places. We use the same strategy for defining static slicing for algebraic Petri nets as proposed by Khan et al in [10]. The major difference between their and our slicing construction is that we use the *neutral transition* together with *reading transition* to reduce the slice size (as discussed

in the section 3). We also introduce a notion of dynamic slicing for the first time in the context of APNs.

5 Case Study

We took a small case study from the domain of crisis management systems (car crash management system) for the experimental investigation of the proposed slicing algorithms. In a car crash management system (CCMS); reports on a car crash are received and validated, and a **Superobserver** (i.e., an emergency response team) is assigned to manage each crash.

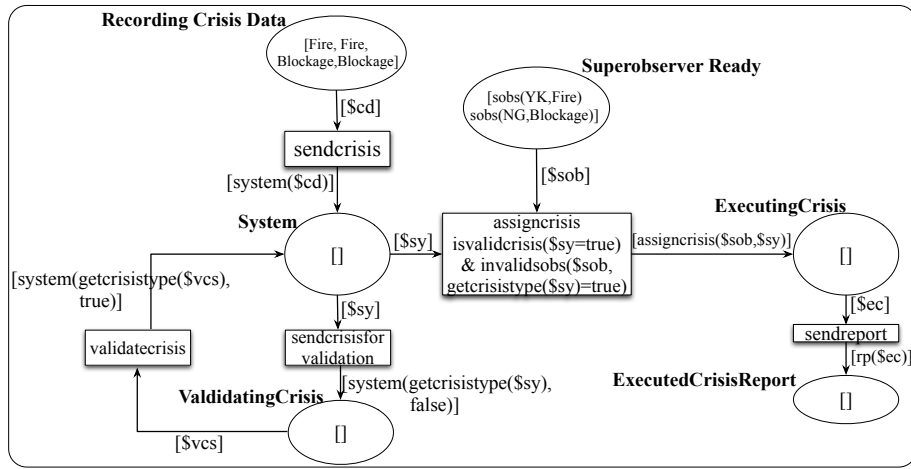


Fig. 8. Car crash APN model

The APN Model can be observed in Fig. 8, it represents the semantics of the operation of a car crash management system. This behavioral model contains labeled places and transitions. There are two tokens in the place **Recording Crisis Data** that are **Fire** and **Blockage**. These tokens are used to mention which type of data has been recorded. The input arc of transition **sendcrisis** takes the **cd** variable as an input from the place **Recording Crisis Data** and the output arc contains term **system(cd)** of sort **sys** (It is important to note that for better readability, we omit **\$** symbol from the terms over the arcs). The **sendcrisis** transition passes a recorded crisis to system for further operations. All the recorded crises are sent for validation through **sendcrisisforvalidation** transitions. Initially, every recoded crisis is set to false. The output arc of **validatecrisis** contains the **system(getcrisistype(vcs), true)** term which sends validated crisis to system. The transition **assignncrisis** has two guards, the first one is **invalid(sy)=true** that enables to block invalid crisis reporting to be executed for the mission and the second one is **invalid(sob, getcrisistype(sy))=**

true which is used to block invalid **Superobserver** (a skilled person for handling crisis situation) to execute the crisis mission. The **Superobserver** YK will be assigned to handle **Fire** situation only. The transition **assigncrisis** contains two input arcs with **sob** and **sy** variables and the output arc contains term **assigncrisis(sob,sy)** of sort **crisis**. The output arc of transition **sendreport** contains term **rp(ec)**. This enables to send a report about the executed crisis mission. We refer the interested reader to [6] for the algebraic specification of a car crash management system.

An important safety threat, which we will take into an account in this case study is that the invalid crisis reporting can be hazardous. The invalid crisis reporting is the situation that results from a wrongly reported crisis. The execution of a crisis mission based on the wrong reporting can waste both human and physical resources. In principle, it is essential to validate a crisis that it is reported correctly. Another, important threat could be to see the number of superobservers should not exceed from a certain limit. Informally, we can define the properties:

Formally we can specify the properties as, let **Crises** be a set representing recorded crisis in car crash management system. Let $isvalid : Crises \rightarrow BOOL$, is a function used to validate the recorded crisis.

$$\varphi_1 = \mathbf{AF}(\forall crisis \in System | isvalid(crisis) = true).$$

$$\varphi_2 = \mathbf{AG}(|SuperobserverReady| \leq 2).$$

In contrast to generate the full state space for the verification of the properties φ_1 and φ_2 , we alleviate the state space by applying our proposed algorithm i.e., *abstract slicing algorithm*. For φ_1 and φ_2 , the criterion places are **System** and **Superobserver Ready**. The unfolded car crash APN model is shown in the Fig. 9. The abstract slicing algorithm takes *an unfolded car crash APN model* and *System (an input criterion place)* as an input and iteratively builds the sliced net for φ_1 . Respectively for φ_2 , the algorithm starts from *Superobserver Ready (as input criterion place)* and builds the slice. The sliced unfolded car crash APN models are shown in the Fig. 10, for the both prperties i.e., φ_1 and φ_2 .

Let us compare the number of states required to verify the given property without slicing and after applying abstract slicing. In the first column of Table.2, the number of states are given that are required to verify the property without slicing and in the second column the number of states are given to verify the property by slicing.

Let us take a criterion place (i.e, *System*) from the car crash APN model and apply our proposed *concerned slicing algorithm* to find which transitions and places can contribute tokens to that place. It is important to note that, we perform concerned slicing directly on the car crash APN model instead of the unfolded car crash APN model (as discussed in the section 3). The sliced car crash APN-model can be observed in the Fig.11.

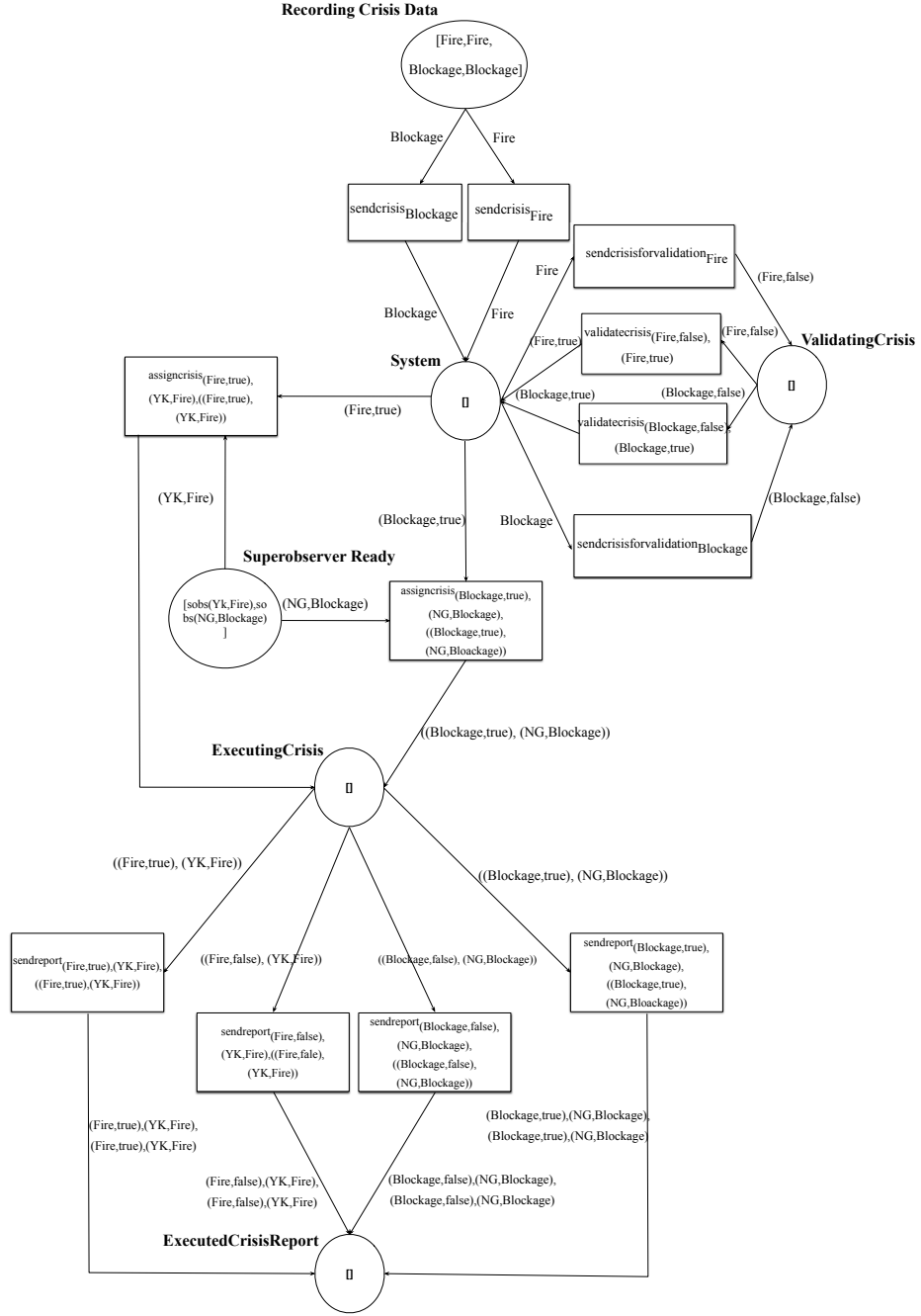


Fig. 9. The unfolded car crash APN model

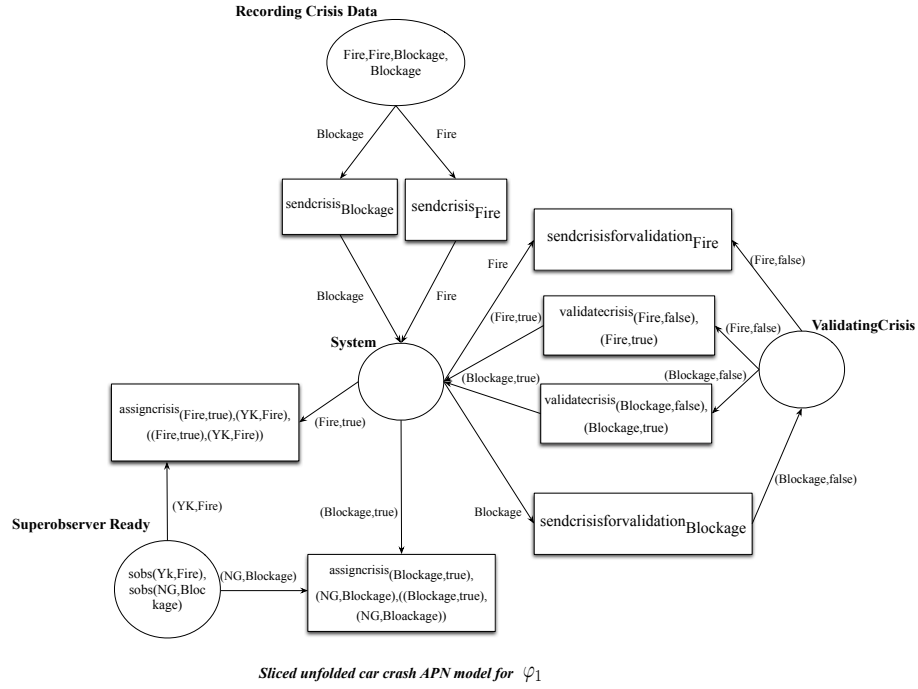
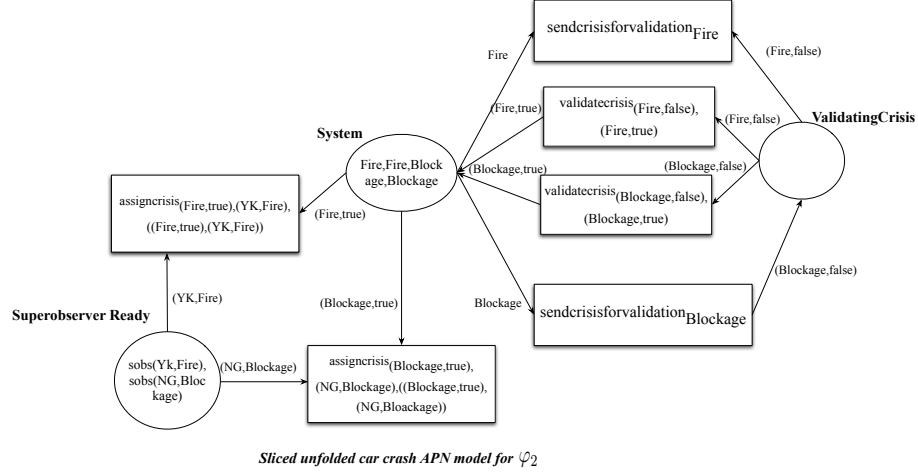


Fig. 10. Sliced unfolded car crash APN model (by applying *Abstract slicing*)

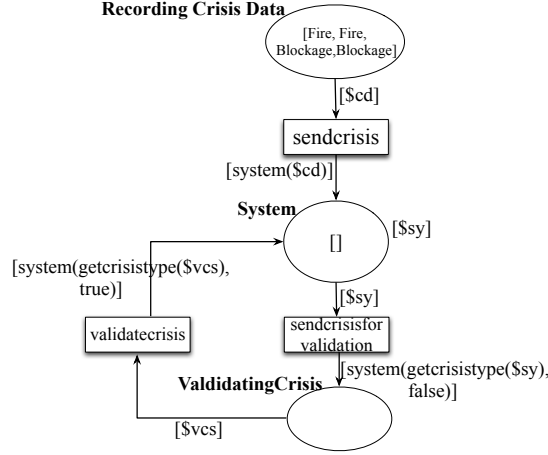


Fig. 11. Sliced car crash APN model (by applying *concerned slicing*)

6 Evaluation

In this section, we evaluate our abstract slicing algorithm and compare with existing slicing construction for APNs (Note: We do not include concerned slicing algorithm in the evaluation. As discussed in section 3, concerned slicing algorithm is designed to improve the testing of APN for the first time. We only include slicing algorithm that are designed to improve the model checking). We measure the effect of slicing in terms of savings of the reachable state space, as the size of the state space usually has a strong impact on time and space needed for model checking.

To show that state space could be reduced for practically relevant properties. We took some specific examples of temporal properties from the different case studies. Instead of presenting properties for which our method the best one, it is interesting to see where it gives an average or worst case results. Let us specify the temporal properties that we are interested to verify on the given APN model. (Note: we refer the interested reader to [8] for APN models of case studies used in the evaluation).

For the *Daily Routine of two Employees and Boss APN model*, for example, we are interested to verify that: “*Boss has always meeting*”. Formally, we can specify the property:

$$\varphi_1 = \mathbf{AG}(NM \neq \emptyset), \text{ where “NM” represents a place not meeting.}$$

For *Simple Protocol*, for example, we are interested to verify that: “*All the packets are transmitted eventually*”. Formally, we can specify the property:

$$\varphi_2 = \mathbf{AF}(|PackTorec| = |PackTosend|), \text{ where “PackTosend and PackTorec” represents places.}$$

And for a *Complaint Handling APN model*, we are interested to verify: “All the registered complaints are collected eventually”. Formally, we can specify the property:

$\varphi_3 = \mathbf{AG}(\text{RecComp} \Rightarrow \mathbf{AFCompReg})$, where “RecComp” (resp. CompReg) means “place RecComp (resp. CompReg) is not empty”.

For an *Insurance claim APN model* an interesting property could be to verify that: “Every accepted claim is settled”. Formally, we can specify the property:

$\varphi_4 = \mathbf{AG}(AC \Rightarrow \mathbf{AFCS})$, where “AC” (resp. CS) means “place AC (resp. CS) is not empty”.

For a *Customer support production system* an interesting property could be to verify that: “Number of requests are always less than 10”. Formally, we can specify the property:

$\varphi_5 = \mathbf{AG}(|\text{Requests}| < 10)$.

For a *Producer Consumer APN model* an interesting property could be to verify that: “Buffer place is never empty”. Formally, we can specify the property:

$\varphi_6 = \mathbf{AG}(|\text{Buffer}| > 0)$.

Table 2. Results with different properties concerning to APN models

<i>System</i>	<i>Property</i>	<i>Tot.States</i>	<i>APNslicing</i>	<i>AbstractSlicing</i>	<i>Reduction</i>
<i>Daily Routine of 2 Employees & Boss</i>	φ_1	80	5	3	96.25%
<i>Simple Protocol</i>	φ_2	21	21	9	57.143%
<i>Complaint Handling</i>	φ_3	2200	679	112	94.91%
<i>A Customer support Production system</i>	φ_4	471	171	91	80.68%
<i>Insurance Claim</i>	φ_5	889	121	49	94.48%
<i>Producer Consumer</i>	φ_6	372	372	372	0.0%

Let us study the results summarized in the table shown in Table. 2, the first column represents the system under observation whereas the second column refers to the property that we are interested to verify. In the third column, total number of states is given based on the initial markings of places. In the fourth column, number of states are given that are required to verify the given property by applying *APNslicing*. In the fourth column, number of states that are required to verify the given property by applying *abstract slicing*. The last column represents the number of states that are reduced (in percentage) after applying Abstract slicing algorithm.

We can draw the following conclusions from the evaluation results:

- *Abstract slicing* often reduces the slice size as compared to *APNslicing* slice size. This is due to the inclusion of *neutral transition* together with *reading transitions*. As a result number of states are reduced to verify the given property, which is an improvement towards model checking. We can observe Table. 2, a part for property φ_2 , there is always an improvement in the reduction of states. It is important to note that at worst the slice size obtained after applying *abstract slicing* is equal to the slice size obtained by applying *APNslicing*.
- Reduction can vary with respect to the net structure and markings of the places (this is true for both *abstract slicing* and *APNslicing*). The slicing refers to the part of a net that concerns to the property, remaining part may have more places and transitions that increase the overall number of states. If slicing removes parts of the net that expose highly concurrent behavior, the savings may be huge and if the slicing removes dead parts of the net, in which transitions are never enabled then there is no effect on the state space.
- It has been empirically proved that in general slicing produces best results for work-flow nets in [10, 16]. Our experiments also prove that for work-flow nets abstract slicing produces better results.
- *Abstract slicing algorithm* is a linear time complex.

7 Conclusion and Future Work

In this work, we have presented two slicing algorithms (i.e., *Abstract slicing* and *Concerned slicing*) to improve the verification of systems modeled in the Algebraic Petri nets. The *Abstract slicing* algorithm has been designed to improve the model checking whereas the *Concerned slicing* has been designed to improve the testing of APNs. Both the algorithms are linear time complex and significantly improves the model checking and testing of APNs.

As a future work, we are targeting to define more refined slicing constructions in the context of APNs and to implement a tool named SLAPn (i.e., slicing algebraic Petri nets). The objective of SLAPn is to show the practical usability of slicing by implementing the proposed slicing algorithms. The initial strategy to implement SLAPn is to extend the AlPiNA (Algebraic Petri net analyzer) a symbolic model checker. As discussed in the section 3, we are using the same unfolding approach as AlPiNA. Certainly, this will help to reduce the implementation effort.

References

1. D. Buchs, S. Hostettler, A. Marechal, and M. Risoldi. Alpina: A symbolic model checker. In J. Lilius and W. Penczek, editors, *Applications and Theory of Petri Nets*, volume 6128 of *Lecture Notes in Computer Science*, pages 287–296. Springer Berlin Heidelberg, 2010.

2. J. R. Burch, E. Clarke, K. L. McMillan, D. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on*, pages 428–439, 1990.
3. J. Chang and D. J. Richardson. Static and dynamic specification slicing. In *Proceedings of the Fourth Irvine Software Symposium*, 1994.
4. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.
5. K. Jensen. Coloured petri nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, volume 254 of *Lecture Notes in Computer Science*, pages 248–299. Springer Berlin Heidelberg, 1987.
6. Y. I. Khan. A formal approach for engineering resilient car crash management system. Technical Report TR-LASSY-12-05, University of Luxembourg, 2012.
7. 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.
8. Y. I. Khan. Optimizing algebraic petri net model checking by slicing. Technical Report TR-LASSY-13-02, University of Luxembourg, 2013.
9. Y. I. Khan. Slicing high-level petri nets. Technical Report TR-LASSY-14-03, University of Luxembourg, 2014.
10. 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.
11. L. Lamport. What good is temporal logic. *Information processing*, 83:657–668, 1983.
12. 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.
13. 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.
14. 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.
15. A. Rakow. *Slicing and Reduction Techniques for Model Checking Petri Nets*. PhD thesis, University of Oldenburg, 2011.
16. 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.
17. W. Reisig. Petri nets and algebraic specifications. *Theor. Comput. Sci.*, 80(1):1–34, 1991.
18. K. Schmidt. T-invariants of algebraic petri nets. *Informatik-Bericht*, 1994.
19. F. Tip. A survey of program slicing techniques. *JOURNAL OF PROGRAMMING LANGUAGES*, 3:121–189, 1995.
20. A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets*, pages 429–528, London, UK, UK, 1998. Springer-Verlag.
21. Y. Wangyang, Y. Chungang, D. Zhijun, and F. Xianwen. Extended and improved slicing technologies for petri nets. *High Technology Letters*, 19(1), 2013.

22. 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.
23. 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.

8 Acknowledgement

This work has been supported by the National Research Fund, Luxembourg, Project RESIsTANT, ref.PHD-MARP-10.