PhD-FSTC-2015-15
The Faculty of Sciences, Technology and Communication

DISSERTATION

Defense held on 01/04/2015 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN *INFORMATIQUE*

by

YASIR IMTIAZ KHAN
Born on 04 September 1983 in Depalpur, (Pakistan)

PROPERTY BASED MODEL CHECKING OF STRUCTURALLY EVOLVING ALGEBRAIC PETRI NETS

Dissertation defense committee

Dr Nicolas Guelfi, dissertation supervisor
Professor, Université du Luxembourg

Dr Didier Buchs
Software Modeling and Verification Group , Geneve
Professor, Université du Geneve

Dr Pascal Bouvry, Chairman
Professor, Université du Luxembourg

Dr Matteo Risoldi
Golden Tech S.A., Switzerland

Dr Raymond Bisdorff, Vice Chairman
Professor, Université du Luxembourg

Acknowledgements

In the name of Allah the most gracious the most merciful

First of all , I would like to thank the almighty Allah (Subhan Wa Taalah) for His countless blessings on me. He blessed me with an exciting opportunity to pursue my career.

Regarding my dissertation, I would like to express my gratitude towards my thesis supervisor, Prof. Nicolas Guelfi. I would never have been able to finish my dissertation without his guidance. The good thing about him is that he never spoon-fed me with the ideas and solutions. He always pushed me to think out of the box and trained me to work independently. I would also like to thank my thesis defense committee members, Prof. Pascal Bouvry, Prof. Raymond Bisdorff. Besides my advisor, I would like to thank the Prof. Didier Buchs and his team members for their collaboration and insightful comments on my work. My sincere thanks also goes to Dr. Matteo Risoldi and Dr. Levi Lucio for guiding my research for the past several years and helping me to develop my background in the field of software engineering. I gratefully acknowledge the funding sources that made my Ph.D. work possible. I was funded by the FNR (Fonds National de la Recherche) Luxembourg for four years.

A special thanks goes to my family members. Words cannot express my feelings especially about my parents. They laugh when I laugh, they cry when I cry. I am thousands of miles away from my mother but whenever I am troubled she gets to know, I wonder how she does that. I am grateful to my father's countless prayers and efforts for me. My siblings are the most valuable assets for me. My big brother who always took care of me, he played father's role in my childhood and became friend when I grew up. I would not have achieved any thing without my close sisters Bushra & Fatimah. The prayers of my sisters for me were what sustained me thus far. I would like to thank my brother in laws especially Mr. Akram Khan who showed me the right path to pursue my career.

Friends are the greatest gift of life and I am lucky to have them (Saleem khan, Abid, Azam, Manazar, Arshad, Mehfooz, Zaheer, Hasnain, Noman, Javaid, Umer). I would like to thank them for their support and company. They are always kind to bear my temper and get along with me.

Finally, and most importantly, I would like to thank my wife Dr. Sidra Tulmuntiha. Her tolerance of my occasional rough moods is a testament in itself of her unyielding devotion and love.

This thesis is dedicated to the people of Depalpur city and our upcoming baby.

Abstract

There are two important challenges in any system development life cycle, the first is to ensure the correctness of a model at the earliest stage possible and the second is to ensure its correctness once it evolves with respect to the emergence of new requirements, performance may need to be improve, business environment is changing. Usual verification techniques such as testing and simulation are used commonly for the verification of a model. The downsides of these techniques are that there is no guarantee of the absence of errors and they need to be repeated after every evolved version.

Model checking is an automatic technique for verifying finite state system models. Although, model checking is proved to be a useful technique, the typical drawback of model checking is its limits with respect to the state space explosion problem. As system gets reasonably complex, completely enumerating their states demands increasing amount of resources. Various techniques like symbolic model checking, on the fly model checking and compositional reasoning partially overcome this problem.

Petri net is a well-known low-level formalism for modeling and verifying concurrent and distributed systems. The modeling of systems by low-level Petri nets is tedious and therefore various advancements have been created to raise the level of abstraction of Petri nets. Among others, Algebraic Petri nets raise the level of abstraction of Petri nets by replacing black tokens with the elements of user defined data types i.e., algebraic abstract data types.

The first contribution of this thesis is to develop an approach to tackle the state space explosion problem for model checking of Algebraic Petri nets by re-using, adapting and refining state of the art techniques. The proposed approach is based on slicing and the central idea is to perform verification only on those parts that may affect the property the Algebraic Petri net model is analyzed for. We propose several slicing algorithms for Algebraic Petri nets and can be applied to Petri nets by slight modifications. The proposed slicing algorithms can alleviate the state space even for certain strongly connected nets and are proved not to increase the state space.

The second contribution is an approach to improving re-verification of structurally evolving Algebraic Petri nets. The idea is to classify evolutions and properties to identify which evolutions require re-verification. We argue that for the class of evolutions that require verification, instead of verifying the whole system, only a part that is concerned by the property can be sufficient.

The third contribution is the development of a stand-alone tool i.e., SLAP$_n$. The objective of this tool is to implement proposed slicing algorithms and to show the practical usability of slicing technique. An interesting exploitation of SLAP$_n$ is that it can be added to any existing model checker as a pre-processing step.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Seek knowledge from the cradle to the grave.

— Muhammad (PBUH)

Software evolution is inevitable in the field of Information and Communication Technology (ICT). Existing software systems continue to evolve due to many reasons such as the impact of system's environment changes, emergence of new requirements due to business changes [Som06]. In general, large organizations prefer to evolve existing software rather than developing new software. There are several keywords, which speak about the evolution such as change, adaptation, variation, modification, transformation etc. It has been shown by statistical evidence that the more a system evolves the more its complexity grows and its performance decreases [Sim03, Lap05, Erl00].

A lot of research work has been going on to develop tools and techniques for the better management of software development and its evolution since last decades. *Iterative refinements and incremental developments* is a commonly used technique for handling complex systems in hardware and software engineering. This involves creating a new specification or implementation by modifying an existing one [LB03, KR12]. In general, the modelers provide a first model that satisfies a set of initial requirements. Then the model can undergo several iterations or refinements until all the expected requirements are satisfied. A model evolves during its life cycle and it is highly desirable for the developer to control the model quality as it evolves.

The problem with the iterative and incremental development is that there is no guarantee that after each iteration or evolution of the model, it will still satisfy the properties it was previously satisfying.

A naive solution is to repeat verification after every evolution to determine whether a system satisfies previously satisfied properties or not (shown in Fig.1.1). The process of establishing correctness of previously satisfied properties by means of verification and re-verification is very expensive (i.e., in terms of time and space) and needs better solution for its management. In the last decades, an attempt to improve

the re-verification has been made by rule-based refinements (also termed as property preserving evolutions). The advantage of this approach is that by construction it is guaranteed that the evolved system model is correct and there is no need to repeat the verification. The disadvantage is that it is extremely difficult to find appropriate evolutions that preserve the previously satisfied properties and fulfills the evolution requirements. Therefore, it is impractical to adapt rule-based refinements.

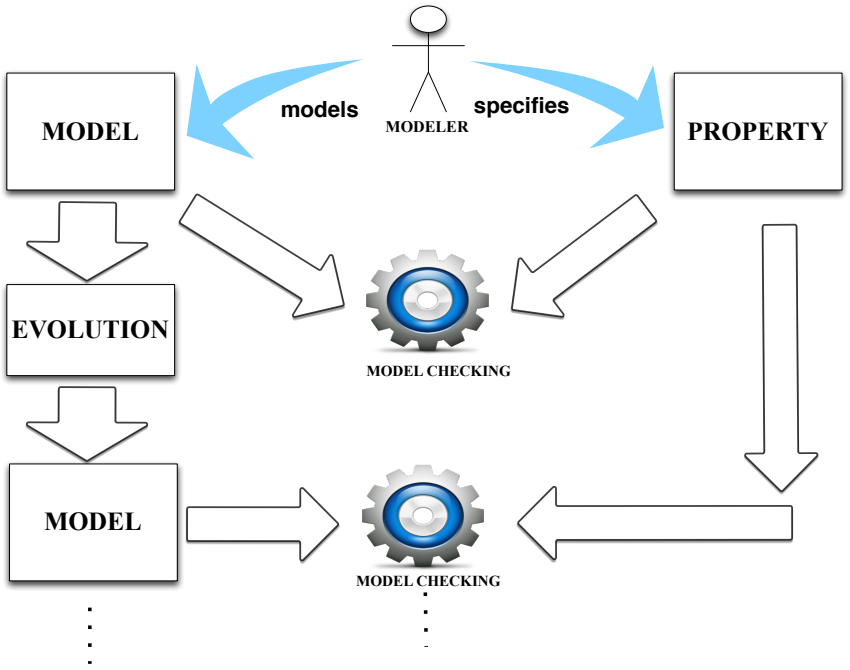


Figure 1.1: A naive Approach

Concurrent and distributed systems are used to connect the computer systems with the real world. These systems are widely used in applications from television sets to train signaling and work-flow systems. The complexity of such systems increases because of the order in which events occur in the their execution is unpredictable and only restricted by synchronization of individual processes. In general, the design of concurrent and distributed systems is complex and there are high possibilities that subtle modeling faults will cause erroneous models (i.e., failing in satisfying desired model properties). Formal modeling is required to manage the complexity of such systems. Many formal modeling languages has been proposed such as *statecharts, Petri nets, X-machines etc.* [Gen87, Hoa78, Hol97, Pet62, Mur89].

Petri nets (PNs) are well known low-level formalism for modeling and verifying concurrent and distributed systems [Pet62]. Petri nets are well suitable to represent in a natural way logical interactions among parts or activities in a system as compared to other popular techniques of graphical systems such as block diagrams or logical trees. Petri nets formalism combine a well defined mathematical theory with a graphical representation of the dynamic behavior of systems. There are two major classes of Petri nets i.e., low-level Petri nets and high-level Petri nets. The drawback of low-

level PN formalism is their inability to represent complex data, which influences the behavior of a system. Various advancements of low-level PN have been created to raise the level of abstraction of PN and are termed as High-Level Petri nets (HLPNs). Among others, APNs raise the level of abstraction of PN by using complex structured data [Rei91, BBG01]. However, APNs can be unfolded into a behaviorally equivalent PNs.

1.1 Formal verification

Formal verification of a software or hardware system is the process of checking whether a system satisfies some properties. There are three basic steps for a formal verification process. The first step is to model a system formally and then formalize the specification as a second step and lastly prove the satisfaction of the specified properties by the formal model. Formal verification helps to prove the correctness of sequential, distributed and concurrent systems models. Several dedicated formal verification approaches are developed such as *theorem proving, equivalence checking, model checking, model based testing*. Theorem proving is a powerful technique that can deal with the infinite states space. In theorem proving, both the specifications and systems are expressed as formulas in some mathematical logic. If a proof can be constructed for the specifications from the axioms of the system, it is claimed that systems satisfies given specifications. The axiom of the system is a starting point of reasoning to be accepted as true without controversy. Equivalence checking is the most common technique used in industry. It uses canonical representations for the comparison such as satisfiability solvers or Binary Decision Diagram (BDD) .

1.1.1 Model Checking

Model checking is a formal approach to automatically verify software or hardware systems [CES86]. Informally, an abstraction of a software or hardware system is designed as a model and properties are designed to know what the system should do. In general, properties are written in propositional temporal logic. By exploring all possible states of the model, it is determined whether the system satisfies given properties or not. In the case when a model does not satisfy its given properties, a report of diagnostic counter examples is produced (shown in Fig.3.1). In the original proposal of model checking [CE82], model refers to the Kripke structure [Kri63] that represents the possible behaviors of the system. Kripke structure is considered as a low-level formalism and from the modeling perspective Kripke structure is not well suitable due to their rapid intractability. To overcome this limitation, an appropriate formalism (such as *Petri nets, X-Machines, UML state charts*) is used to model a system.

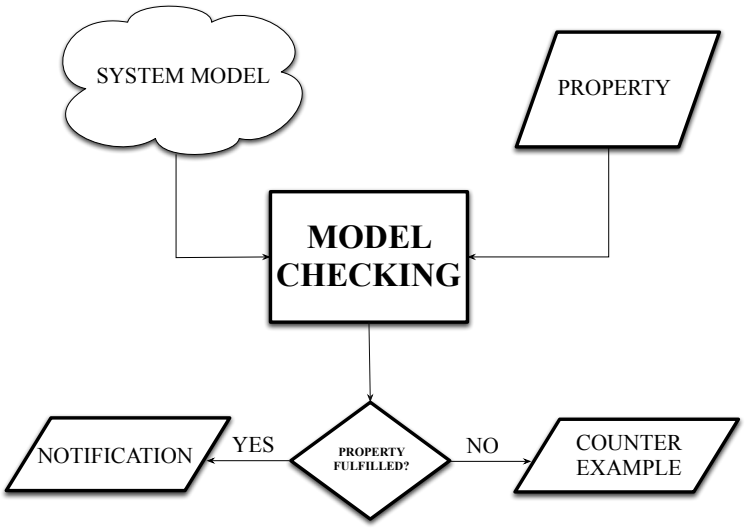


Figure 1.2: Model checking

1.1.2 Improving Model checking

Although model checking has been proved to be a useful technique it still has many issues such as the famous 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 [Val98, Lam83]. It is also important to note that a designer must make sure that the model and properties are conforming to the customer expectations. Otherwise, the results of model checking are useless. 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 [BHMR10, BCM+90].

This thesis identifies and investigates fundamental questions related to model checking and repeated model checking of software evolution such as:

Is it possible to use properties to reduce the state space generated for model checking?

Is it possible to reduce the re-verification cost of an evolved model by restricting the model checking to the model part that evolved?

This thesis is addressing these two questions in the context of models given in terms

of Algebraic Petri nets and properties specified using temporal logic such as Linear temporal logic (LTL) or Computation tree logic (CTL).

1.2 Contributions

The primary focus of this thesis is to improve the verification and re-verification of systems modeled in Petri nets. We divide our contributions as follows:

1.2.1 Property based model checking of Algebraic Petri nets

The first contribution of this work is to propose an approach for property-based reduction of the state space of Algebraic Petri Nets (APNs). Reduction methods such as Petri nets Slicing (PN Slicing) are used to improve the model checking by syntactically reducing PN model. 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. As shown in Fig.1.3, a PN model is sliced with respect to different places (i.e., *B and C*). These slices constitute only the portion of the model that impacts the property to be verified. Typically, these slices will have a smaller state space, thus reducing the cost of model checking. Another interesting application of PN Slicing is its to use to improve testing. The idea is to generate sliced net with respect the desired set of places or transitions and generate test input data for only the sliced net. It is important to note that construction methodology of slicing algorithms designed for testing is different from the slicing algorithms that are designed to improve model checking.

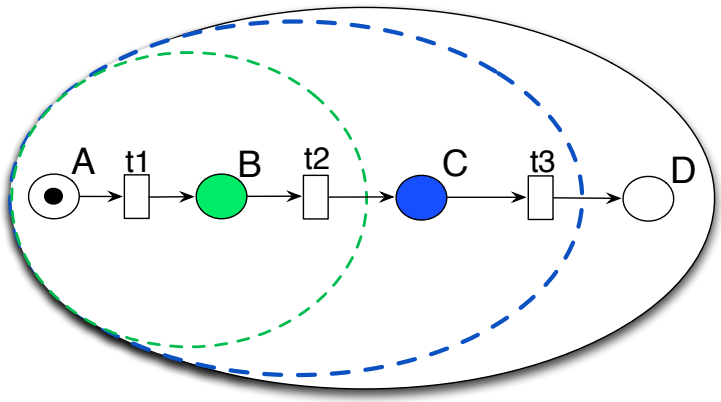


Figure 1.3: PN model and its slices w.r.t places *B and C*

One limitation of PN Slicing is that all the slicing algorithms are limited to Petri nets. To the best of our knowledge there is no proposal for slicing APNs. One characteristic of APNs that makes them complex to slice is the use of multisets of algebraic terms over the arcs. In principle, algebraic terms may contain variables. Even though, we want to reach a syntactically reduced net (to be semantically valid), its reduction by slicing, needs to determine the possible ground substitutions of these algebraic terms. We use partial unfolding proposed in [BHMR10] to determine ground substitutions of the algebraic terms over the arcs of an APN.

We developed several slicing algorithms for APNs [KR13, Kha14, KG14b]. These algorithms are designed to improve model checking and testing of APNs. Although, these algoirthms can be applied to other classes of PNs with slight modifications. In general, slicing algorithms designed in the context of PNs struggles with the strongly connected nets and do not generate smaller slice. The more reduced model means more reduction in the state space. The general idea of slicing APNs is shown in Fig. 1.4, at first an APN model is partially unfolded and criterion places are extracted from the temporal description of properties. Then by applying a slicing algorithm, sliced unfolded net is obtained. This sliced net is used to verify the given property instead of the original APN model.

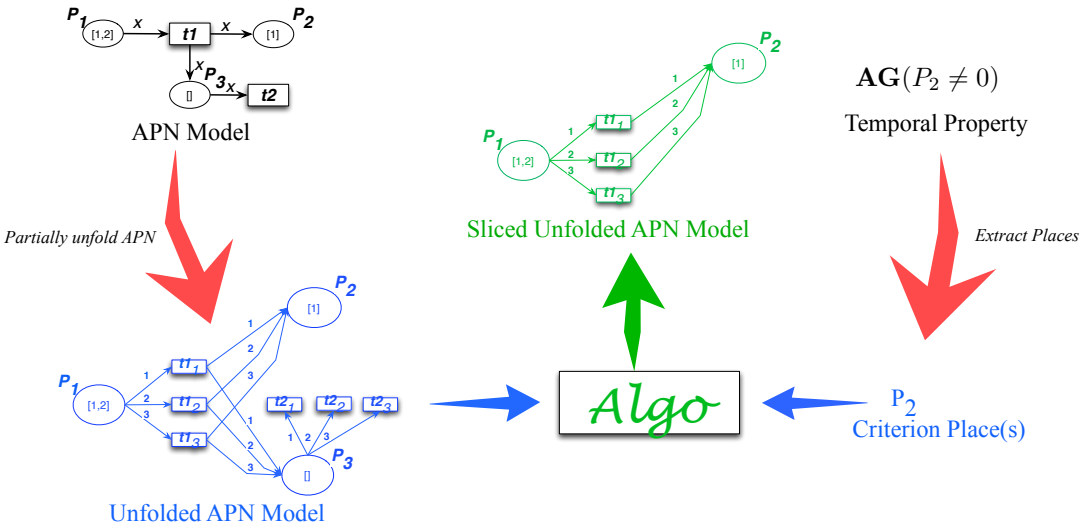


Figure 1.4: APNs slicing overview

Our proposed slicing algorithms can alleviate the state space even for certain strongly connected nets and are proved not to increase the state space. The resultant sliced APN model allows verification and falsification if it is slice fair. In the first column of Table1.1, our proposed APN slicing algorithms are shown [Kha13b]. The second column represents the class of properties that are preserved by the algorithm. Depending on the construction methodology, slicing algorithms preserves certain specific properties

It is important to note that all the proposed APN slicing algorithms can be easily

Table 1.1: Proposed Slicing Algorithms for Algebraic Petri nets

Algorithm	*Preserved Prop*	*Type Slicing*
APN Slicing	LTL_{-X}	*Static*
Abstract Slicing	CTL^*_{-X}	*Static*
Safety Slicing	*Safety*	*Static*
Liveness Slicing	*Livenss*	*Static*
Concerned Slicing	*Particular*	*Dynamic*
Smart Slicing	*Particular*	*Dynamic*

adapted to be applied to low-level PN, thus introducing new slicing techniques for PNs.

1.2.2 Property based model checking of structurally evolving APNs

The second contribution of this work is an approach improving the verification of structurally evolving APNs. The proposal is based on slicing and on the classification of evolutions and properties. Informally, APNs consist of a control part, which is handled by a PN and a data part, which is handled by one or more Algebraic Data Type (ADT). The control part of an APN can evolve by *add/remove places, transitions, tokens and terms over the arcs.* However, the data part can evolve by changing the types of variables, operations used in ADT. Our contribution focuses on the control part of the model and does not cover evolutions of data part.

Our proposal pursues three main goals. The first is to perform verification only on those parts that may influence the property satisfaction for the analyzed APN model. The second is to classify the evolutions and properties to identify which evolutions require re-verification. We show that for the class of evolutions that require verification, instead of verifying the whole system only a part that is concerned by the property would be sufficient. The third goal is closely related to the previous proposals of property preserving evolutions. Padberg and several other authors proposed rule based modifications for the invariant preservation of APNs[PGE98, LLV12, Er97]. The theory of rule based modification is an instance of High-level Replacement System (HLRS) and preserving properties (i.e., either structural or behavioral properties) by preserving morphisms. Our proposal of slicing further improves the previous proposals of property preservation. The idea is to preserve morphisms restricted to the sliced part of the model[Kha13a].

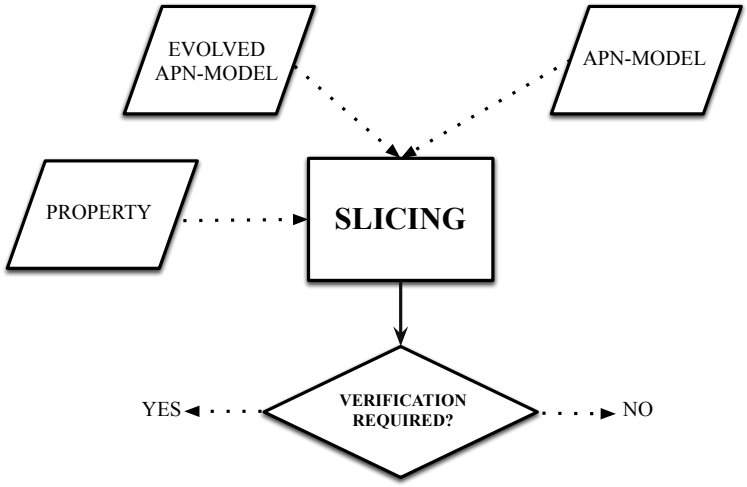


Figure 1.5: Property based Model checking of structurally evolving APNs

As shown in the Fig.1.5, slices are generated for APN model and its evolved APN model. Then by comparing both APN models and classification of evolutions, it is determined if we require re-verification or not.

1.2.3 SLAP$_n$: A tool for slicing PNs and APNs

Third contribution of this work is the development of a stand alone tool (i.e., SLAP$_n$) for slicing PNs and APNs [KG14a]. To the best of our knowledge this is the first effort to develop a stand alone slicing tool. The objective of SLAP$_n$ is to show the practical usability of slicing by implementing the proposed slicing algorithms.

The tool facilitates a user to draw an APN/PN model using a graphical editor and write the temporal description of a property (shown in Fig.1.6). From the palette a slicing algorithm can be selected. We have implemented various slicing algorithms such as *APNSlicing, Abstract slicing, Safety slicing, Liveness slicing, Concerned slicing, Smart slicing*. The tool automatically extracts places from the property description and a sliced net model is produced. It is important to note that SLAP$_n$ itself is not a model checker. Perhaps, It can be integrated with any existing model checker to generate the state space.

1.3 Organization of this thesis

In this chapter, we discuss the motivation and the main contributions of this thesis. The remaining chapters are structured as follows:

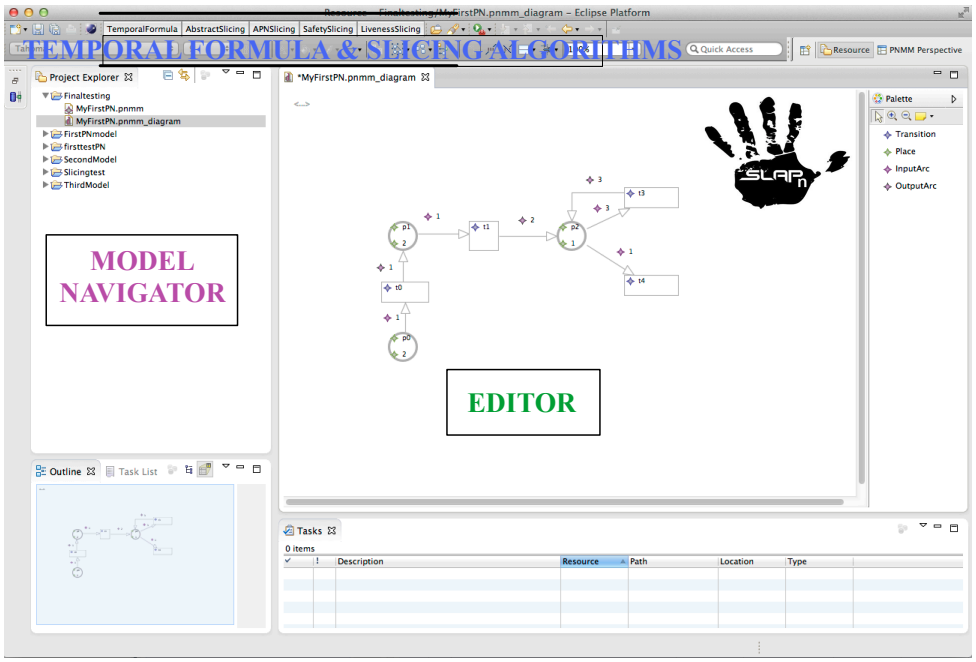


Figure 1.6: SLAP$_n$'s main screen

- **Chapter** 2: This chapter consists of informal and formal descriptions about the modeling formalism used in this thesis. We also use graphical representations when defining different classes of Petri nets such as PNs and APNs. This chapter also includes formal definition of the temporal logic such as LTL/CTL.

- **Chapter** 3: This chapter is about the detailed survey of existing slicing algorithms and their comparison with each other. We highlight the objective of every proposed algorithm together with their informal and formal description. We divide slicing algorithms into two different groups i.e., one used to improve testing and one used to improve model checking process.

- **Chapter** 4: This chapter consists of the detailed underline theory and algorithms about the first contribution (i.e., property based model checking of APNs) of this thesis is given. With the help of extremely simple examples of Algebraic Petri nets, every proposed slicing algorithm is explained.

- **Chapter** 5: The second contribution (i.e., property based model checking of structurally evolving APNs) of this thesis is discussed in this chapter.

- **Chapter** 6: This chapter contains detailed evaluation of our proposed approaches. In the first half, we took a small case study from the domain of crisis management system (Car Crash Management System) to exemplify the proposed approach. In the second half, proposed slicing algorithms are evaluated on the benchmark case studies.

- **Chapter** 7: This chapter consists of a tool description (SLAP$_n$), which is third contribution of this thesis.

- **Chapter** 8: This chapter concludes the proposed work in this thesis and opens new perspectives.

Chapter 2

Informal and Formal Definitions

He who performs not practical work nor makes experiments will never attain to the least degree of mastery.

— Jaber ibn Hayyan

This chapter comprises basic informal and formal definitions of modeling formalism used in this work. (Note, Formal definitions, propositions, lemmas and theorems used in this work are taken as it is or with slight modifications from [Rei91, Sch94, Rak11, Jen87]

2.1 Petri nets Definitions

Petri nets are very well known formalism to model and analyze concurrent and distributed systems. C.A. Petri in his Ph.D. Dissertation introduced Petri nets formalism [Pet62].

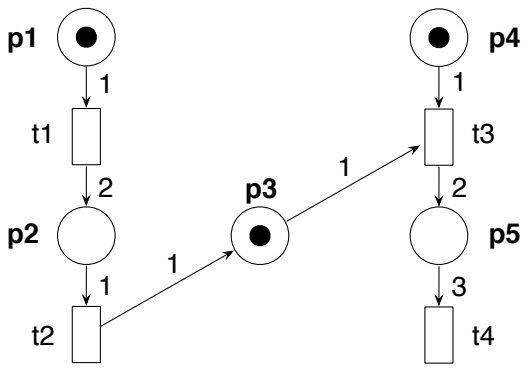


Figure 2.1: Eample Petri net model

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. Formally, we can define PNs:

Definition 2.1.1: Petri Nets
A Petri Net is: $PN = \langle P, T, f, w \rangle$ consist of

- 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 function $w : f \to \mathbb{N}$, assigns weights to the arcs.

An example PN model $\langle P, T, f, w \rangle$ shown in Fig.2.1, is defined as follows. Let W and F be sets representing arcs and weights respectively,

- $P = \{p1, p2, p3, p4, p5\}$
- $T = \{t1, t2, t3, t4\}$
- $F = \{\langle p1, t1\rangle, \langle t1, p2\rangle, \langle p2, t2\rangle, \langle t2, p3\rangle, \langle p3, t3\rangle, \langle p4, t3\rangle, \langle t3, p5\rangle, \langle p5, t4\rangle\}$
- $W = \{\langle p1, t1\rangle \mapsto 1, \langle t1, p2\rangle \mapsto 2, \langle p2, t2\rangle \mapsto 1, \langle t2, p3\rangle \mapsto 1, \langle p3, t3\rangle \mapsto 1, \langle p4, t3\rangle \mapsto 1, \langle t3, p5\rangle \mapsto 2, \langle p5, t4\rangle \mapsto 3\}$

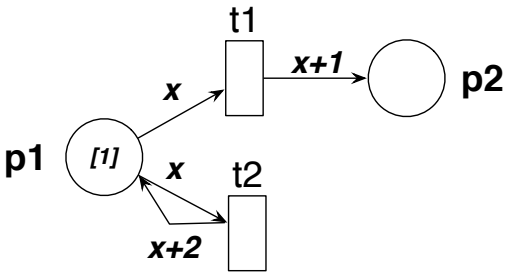
The semantics of a PN expresses the *non-deterministic* firing of transitions in the net. Firing a transition means consuming tokens from a set of places linked by the input arcs of a transition and producing tokens into a set of places linked by the output arcs of a transition. A transition can be fired only if its incoming places have a quantity tokens greater than the weight attached to the arc. As shown in Fig.2.1, transitions *t1 and t2* are enabled from the initial marking and *non-deterministically* any one of them can fire. Let us consider that if *t1* fires, the result of transition firing will remove a token from place *p1* and adds a token to place *p2*.

We can build models of computer systems (or any other system) by means of PNs (Note: The above defined PNs are also termed as low-level PNs). However, PNs lack to represent complex data which influences the behaviour of the system. To overcome this issue, various advancements of PNs have been created such as such as Colord Petri Nets (CPN), Predicate/Transition Nets (Pr/T-nets), APNs etc. (termed as High Level Petri Net (HLPN)). [GL79, Rei91, Jen81, Jen87].

The first successful HLPN were created by Hartmann Genrich and Kurt Lautenbach in 1979 [GL79]. The main idea is to distinguish tokens from each other (i.e., tokens are represented by a color instead of black dots). Depending on the initial markings of places and colors of tokens, transitions can occur in many ways. Guards and ex-

pressions are used to specify enabling and disabling of transitions occurrences. The problem with Pr/T-nets was usage of only one class of token colors. Later, it was resolved by allowing each place to have its own set of possible token colors [Jen81]. Among others, in APNs the level of abstraction of PNs is raised by using complex structured data [Rei91]. APNs has two aspects:

- control aspect, which is handled by a PN.
- data aspect, which is handled by one or many AADT.




```
//The naturals ADT.
Adt nat
   Sorts nat;
Generators

       zero : nat;
       suc : nat -> nat;
Operations

       dec : nat -> nat;
       plus : nat, nat -> nat;
Axioms

//dec

       dec(suc($x)) = $x;
//plus

       plus (zero, $x) = $x;
       plus (suc($x), $y) = suc
(plus ($x,$y));
Variables

       x : nat;
       y : nat;
```

Figure 2.2: Eample APN model and associated AADT

As shown in Fig.2.2, an APN consists of graphical structure and AADT associated with it. There are no more black dots representing the markings of a place. The type of token is of sort *nat* (with the value suc(zero)) and arcs contains algebraic terms. Let us formally define algebraic specifications that are required to define APNs.

Definition 2.1.2: Signature
A signature $\Sigma = (S, OP)$ consists of a set S of sorts and a family $OP = (OP_{w,s})_{w \in S^*, s \in S}$ of operation symbols. For ϵ being the empty word, we call $OP_{\epsilon,s}$ the set of constant symbols.

In the above definition, set of sorts and operations are disjoint, where S^* is a finite sequence of words. For example, if $S = \{nat, bool\}$, then $op_{(nat,nat,bool)}$ represents a binary predicate symbol such as greater than (>) or as an equality =.

Definition 2.1.3: Variable
A set X of Σ-variables is a family $X = (X_s)_{s \in S}$ of variables, disjoint to OP.

For example if $nat \in S$ is a sort then a natural variable would be x_{nat}.

Definition 2.1.4: Terms
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 $T_{OP,s} =_{def} T_{OP,s}(\emptyset)$ contains the ground terms of sort $s, T_{OP}(X) =_{def} \bigcup_{s \in S} T_{OP,s}(X)$ is the set of Σ-terms over X and $T_{OP} \equiv T_{OP}(\emptyset)$ is the set of Σ-ground terms.

Terms are built upon the values and operations of a signature. For example if *nat* is a sort then natural constants, variables and operators of output sort *nat* are terms of sort *nat*.

Definition 2.1.5: Substitution
Let X be a finite set of Σ-variables. A substitution over X is a mapping $sbt : X \rightarrow T_{OP}(X)$, whereby all $x \in X_s$ it holds $sbt(x) \in T_{OP,s}(X)$. If the image of sbt is contained in T_{OP}, sbt is called a ground substitution.

Let $t \in T_{OP,s}(Y)$, X a finite subset of Y and sbt a substitution over X. Then the term $sbt(T)$ results from T by simultaneously replacing the variables $x \in X$ by the corresponding terms $sbt(x)$.

Definition 2.1.6: Sigma Equation
A Σ-equation of sort s over X is a pair (l, r) of terms $l, r \in T_{OP,s}(X)$.

The equations define the meanings and properties of operations in OP.

Definition 2.1.7: Algebraic Specifications
An algebraic specification $SPEC = (\Sigma, E)$ consists of a signature $\Sigma = (S, OP)$ and a set E of Σ-equations.

An algebraic specifications are obtained by combining a signature and a set of sigma equations.

Definition 2.1.8: Sigma Algebra
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.

It deals with the question how to interpret algebraic specifications to a particular domain.

Definition 2.1.9: Assignment
An assignment of Σ-variables X to a Σ-algebra A is a mapping $asg : X \rightarrow A$,with $asg(x) \in A_s$iff $x \in X_s$. asg is canonically extended to $\overline{asg} : T_{OP}(X) \rightarrow A$, inductively defined by

1. $\overline{asg}(x) \equiv asg(x)$ for $x \in X$;

2. $\overline{asg}(c) \equiv N_c$ for $c \in OP_{\epsilon,s}$;

3. $\overline{asg}(op(t_1, \ldots, t_n)) \equiv N_{op}(\overline{asg}(t_1)), \ldots, \overline{asg}(t_n))$ for $op(t_1, \ldots, t_n) \in T_{OP}(X)$.

All the algebraic terms are mapped to domain values of a sigma algebra.

Definition 2.1.10: Equivalence
Let SPEC-algebra is $SPEC = (\Sigma, E)$ 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 $asg : X \rightarrow A, \overline{asg}(t_1) = \overline{asg}(t_2)$.

For algebraic specification two terms are considered equivalent if all of their assignments are same.

Definition 2.1.11: Multiset
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

$\{b \in B \mid ms_B(b) \neq 0\}$ is finite.

A multiset is an unordered collection of items that may contain duplicates and is the basic data type structure used in HLPN. For example, a multiset $B = [1, 1, 3, 3, 2]$ contains duplication of some elements with unoredered collection of elements.

Definition 2.1.12: Addtion and Subtraction of Multisets
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 $ms3_B \in MS_B$. $\forall b \in B, ms3_B(b) = ms1_B(b) + ms2_B(b)$.

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

Different operations can be performed on multisets. Let us take two example multisets such as $A = [a, a, b]$ and $B = [a, b]$. The addition of A and B will result $A + B = [a, a, a, b, b]$ and the subtraction will result $A - B = [a]$.

Definition 2.1.13: Comparison and equivalence of Multisets
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, $ms1_B = ms2_B$.

Similar to the above Def.2.1.12, further operations can be defined on the multisets such as comparison and equivalence. Let us take two example multisets such as $A = [a, a, b]$ and $B = [a, b, a, a, b]$. Multiset A is less than multiset B and noted by $A \leq B$. Whereas both not equal as well and noted by $A \neq B$.

Definition 2.1.14: Algebraic Petri net
A marked Algebraic Petri Net $APN =< SPEC, P, T, f, asg, cond, \lambda, m_0 >$ 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 Σequations 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, $cond : T \rightarrow \mathcal{P}_{fin}(\Sigma - equation)$, 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.

An APN is a mixture of PN and algebraic specifications. There are no more black dots to represent the markings of places. Associated transitions may have guard conditions to control the firing sequences. An example APN can be observed in Fig.2.2. Both places are of sort naturals.

Definition 2.1.15: Preset(resp. Postset) Places
The preset of $p \in P$ is ${}^\bullet p = \{t \in T | (t,p) \in f\}$ and the postset of p is $p^\bullet = \{t \in T | (p,t) \in f\}$. The pre and post sets of $t \in T$ are defined respectively as: ${}^\bullet t = \{p \in P | (p,t) \in f\}$ and $t^\bullet = \{p \in P | (t,p) \in f\}$.

A preset place consists of all the input places to a particular transition, whereas a postset place consists of the output places to a particular transition. We also note $\bullet P$ (resp. $P\bullet$) as a set of all preset (resp. postset) of places. In example APN shown in Fig.2.2, transition $t1$ has $p1$ place in its preset places and $p2$ in its postset places.

Definition 2.1.16: Enabled Transitions
Let m *and* m' two markings of APN and t a transition in T then $< m, t, m' >$ is a valid firing triplet (denoted by $m[t\rangle m'$) iff

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

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

A transition is enabled only if its incoming place contains a multiset of values equal or greater than the one defined by the weight attached to arc. In example APN shown in Fig.2.2, transition $t1$ is enabled.

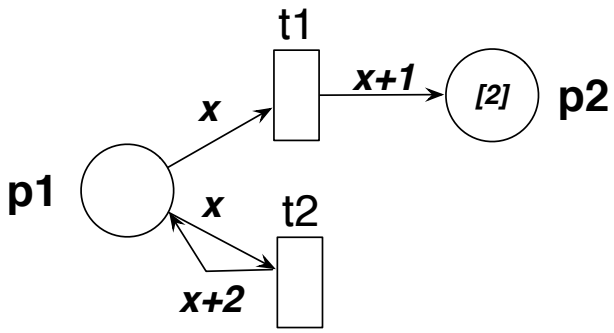


Figure 2.3: Resultant APN after firing transition $t1$

Once a transition is enabled it can fire depending on the guard condition. Firing a transition will remove tokens from the input place and adds tokens to output place. Let us consider the firing of transition $t1$, which is enabled shown in Fig.2.2. The resultant APN can be observed in Fig.2.3. A token valued 1 is removed from place $p1$ and a token with value 2 is added in place $p2$.

Definition 2.1.17: Maximal Firing Sequence
A firing sequence σ of a marked APN is maximal iff $\nexists t \in T : m_0[\sigma t\rangle$, where $|\sigma| \in (\mathbb{N} \cup \{\infty\})$.

Definition 2.1.18: Infinite Firing Sequence
Let $\sigma = t_1, t_2 \ldots$ be an infinite firing sequence of APN with $m_i[t_{i+1}\rangle m_{i+1}, \forall i \in \mathbb{N}$. σ permanently enables $t \in T$ iff $(\exists i \in \mathbb{N})(\forall j \in \mathbb{N})\ i \leq j \Rightarrow m_j[t\rangle$.

Definition 2.1.19: Kripke Structure
$KS = \langle AP, S, L, R, S_0 \rangle$ where:

- AP is a set of atomic propositions,
- S is a set of states,
- $L : S \rightarrow \mathcal{P}(AP)$ is a state labeling function.
- $R \subseteq S \times S$ is a left total transition relation (i.e., $\forall s \in S, \exists s' \mid \langle s, s' \rangle \in R$),
- $S_0 \subseteq S$ is a set of initial states,

Kripke structure are central to model checking as it represents a model of the system under consideration.

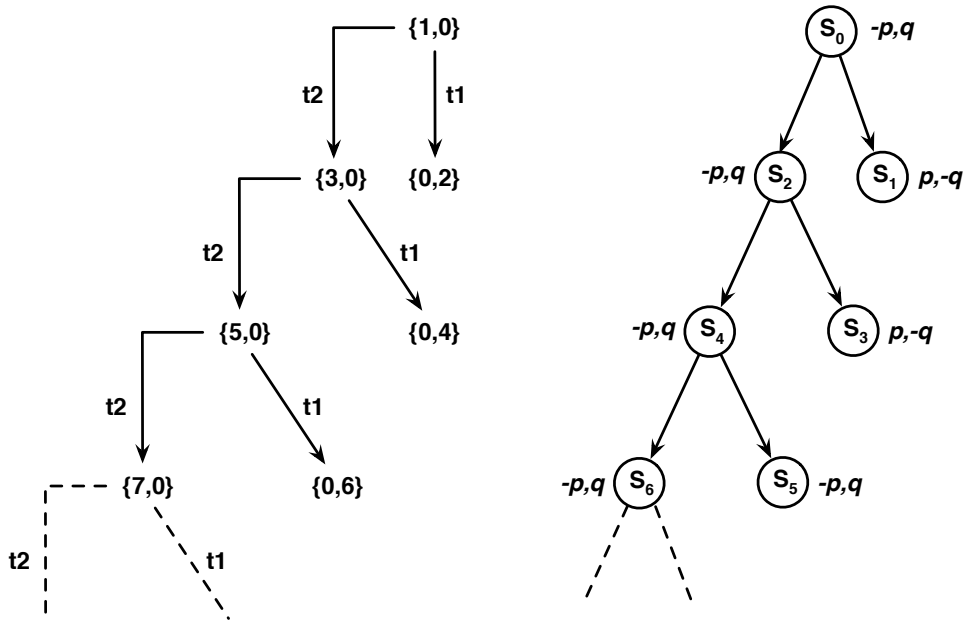


Figure 2.4: Marking graph and Kripke structure of example APN model, Fig.2.2.

As an example of a Kripke structure, let us consider an example APN model shown in the Fig.2.2 and see its corresponding structure in Fig.2.4. Let $AP = \{p, q\}$ be a set of atomic properties where p stands for place p_1 is not empty and respectively q stands for place p_2 is not empty. If any of the place has tokens then it is shown with the negation sign. Let $S = \{s_1, s_2, s_3, ..., s_n\}$ be the set of states the system can take and let the set of initial states be $S_0 = \{s_0\}$.

Definition 2.1.20: Path
Let $KS = \langle AP, S, L, R, S_0 \rangle$ be a Kripke structure. We call path π a sequence of states $\pi = s_0, s_1, s_2, s_3, ...s_i$, such that $\forall i \geq 0, \rangle s_i, s_{i+1} \in R$.

A path refers to the existence of a path to a state that is labeled with a property of interest.

Definition 2.1.21: Reachability Property
Let $KS = \langle AP, S, L, R, S_0 \rangle$ be a Kripke structure. We call that a property $\phi \in AP$ is reachable if there exists a path $\pi = s_0, s_1, s_2, s_3, ...s_i$, and $\exists i \mid \phi \in L(s_i)$.

A state satisfying property is found on a path starting from one of the initial states.

Linear temporal logic (LTL) and computation tree logic (CTL, CTL*) are commonly used for specifying properties of concurrent systems. LTL has been proposed by Amir Pnueli in 1977 [Pnu77], whereas CTL and CTL* invented by E.M. Clarke and E.A. Emerson in 1981 [EH86]. LTL and CTL are concerned only with infinite paths. From here on, π will always denote an infinite path. Furthermore, π_0, π_1 will always denote first and second elements of π, and so on.

Definition 2.1.22: LTL BNF
A well-formed LTL formula, φ, is recursively defined by the BNF formula:

φ ::=> $\top$; top, or true

| $\bot$; *bottom, or false*

| p; p ranges over AP

| $\neg\varphi$; negation

| $\varphi \wedge \varphi$; conjunction

| $\varphi \vee \varphi$; disjunction

| $X\varphi$; next time

| $F\varphi$; eventually

| $G\varphi$; always

| $\varphi U \varphi$; until

The lowercase letters such as p, q, and r, will denote atomic propositions and φ and ψwill denote formulae. CTL*

Definition 2.1.23: LTL Semantics
Let $\models$ denotes a satisfaction relation and the satisfaction is with respect a pair $(\mathbf{M}, \pi)$, a Kripke structure and a path thereof.

$\mathbf{M}, \pi \models \top$; true is always satisfied

$\mathbf{M}, \pi \not\models \bot$; false is never satisfied

$(\mathbf{M}, \pi \models p)$ if and only if $(p \in L(\pi_0))$

$(\mathbf{M}, \pi \models \neg\varphi)$ if and only if $(\mathbf{M}, \pi \not\models \varphi)$

$(\mathbf{M}, \pi \models \varphi \wedge \psi)$ if and only if $(\mathbf{M}, \pi \models \varphi) \wedge (\mathbf{M}, \pi \models \psi)$

$(\mathbf{M}, \pi \models \varphi \vee \psi)$ if and only if $(\mathbf{M}, \pi \models \varphi) \vee (\mathbf{M}, \pi \models \psi)$

$(\mathbf{M}, \pi \models X\varphi)$ if and only if $(\mathbf{M}, \pi^1 \models \varphi)$ next time φ

$(\mathbf{M}, \pi \models F\varphi)$ if and only if $(\exists i \mid \mathbf{M}, \pi^i \models \varphi)$ eventually φ

$(\mathbf{M}, \pi \models G\varphi)$ if and only if $(\forall i \mid \mathbf{M}, \pi^i \models \varphi)$ always φ

$(\mathbf{M}, \pi \models \varphi U \psi)$ if and only if $[\exists i \mid (\forall j < i(\mathbf{M}, \pi^j \models \varphi)) \wedge (\mathbf{M}, \pi^i \models \psi)]$ φ until ψ

Definition 2.1.24: CTL BNF
A well-formed CTL formula, φ, is recursively defined by the BNF formula:

φ ::=> $\top$

| $\bot$

| $p; p$

| $\neg\varphi$

| $\varphi \wedge \varphi$

| $\varphi \vee \varphi$

| $AX\varphi$; A- for all paths

| $AF\varphi$;

| $AG\varphi$;

| $\varphi AU\varphi$

| $EX\varphi$; E- for all paths

| $EF\varphi$;

| $EG\varphi$;

| $\varphi EU\varphi$

Definition 2.1.25: CTL Semantics
Let $\models$ denotes a satisfaction relation and the satisfaction is with respect a pair $(\mathbf{M}, s)$, a Kripke structure and a state thereof.

$\mathbf{M}, s \models \top$

$\mathbf{M}, si \not\models \bot$

$(\mathbf{M}, s \models p)$ if and only if $(p \in L(s))$

$(\mathbf{M}, s \models \neg\varphi)$ if and only if $(\mathbf{M}, s \not\models \varphi)$

$(\mathbf{M}, s \models \varphi \wedge \psi)$ if and only if $(\mathbf{M}, s \models \varphi) \wedge (\mathbf{M}, s \models \psi)$

$(\mathbf{M}, s \models \varphi \vee \psi)$ if and only if $(\mathbf{M}, s \models \varphi) \vee (\mathbf{M}, s \models \psi)$

$(\mathbf{M}, s \models AX\varphi)$ if and only if $(\forall \pi \mid \pi_0 = s, \mathbf{M}, \pi^1 \models \varphi)$ for all paths starting at s, next time φ

$(\mathbf{M}, s \models AF\varphi)$ if and only if $(\forall \pi \mid \pi_0 = s, (\exists i \mid \mathbf{M}, \pi^i \models \varphi))$ for all paths starting at s, eventually φ

$(\mathbf{M}, s \models AG\varphi)$ if and only if $(\forall \pi \mid \pi_0 = s, (\forall i \mid \mathbf{M}, \pi^i \models \varphi))$ for all paths starting at s, always φ

$(\mathbf{M}, s \models \varphi AU \psi)$ if and only if $(\forall \pi \mid \pi_0 = s, (\exists i (\forall j < i \mid \mathbf{M}, \pi^j \models \varphi) \wedge \mathbf{M}, \pi^i \models \psi))$ for all paths starting at s, φ until ψ

$(\mathbf{M}, s \models EX\varphi)$ if and only if $(\exists \pi \mid \pi_0 = s, \mathbf{M}, \pi^1 \models \varphi)$ there exists a path such that next time φ

$(\mathbf{M}, s \models EF\varphi)$ if and only if $(\exists \pi \mid \pi_0 = s, (\exists i \mid \mathbf{M}, \pi^i \models \varphi))$ there exists a path such that eventually φ

$(\mathbf{M}, s \models EG\varphi)$ if and only if $(\exists \pi \mid \pi_0 = s, (\forall i \mid \mathbf{M}, \pi^i \models \varphi))$ there exists a path such that always φ

$(\mathbf{M}, s \models \varphi EU \psi)$ if and only if $(\exists \pi \mid \pi_0 = s, (\exists i (\forall j < i \mid \mathbf{M}, \pi^j \models \varphi) \wedge \mathbf{M}, \pi^i \models \psi))$ there exists a path such that φ until ψ

Chapter 3

Survey of Petri nets Slicing

Knowledge is life, while ignorence is death.

— Ali (RA)

A fundamental challenge in any system development process is to ensure the correctness of the design at the earliest stage possible. Testing and simulation are usual techniques for the validation of a design. The major drawback of these techniques is that there is no guarantee of the absence of errors. Model checking is an automatic technique for verifying finite state systems. It has been proved to be useful alternative to simulation and testing. Any one who can run simulations of a design is capable of model checking the same design.

The process of model checking consists of three tasks as shown in Fig. 3.1. The first task is to design a model in a language accepted by the model checking tool. The second is to specify desired properties in some logical formalism. In general, temporal logic is used to assert how the behavior of the system evolves over the time. Finally, it is shown if a model satisfies the property or not by completely enumerating the state space of a model. In case of negative result a counterexample is produced, which is useful in tracking down where the error has occurred. Model checking is proved to be useful technique but the main challenge is the state space explosion problem.

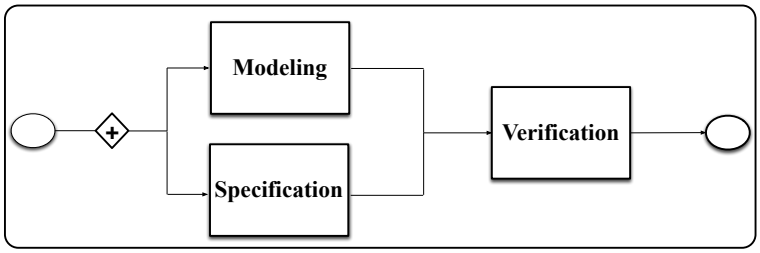


Figure 3.1: Process of model checking

In recent years, the problem of state space explosion is deeply studied and some major improvements have been made [BCM+90, GP93, GL91, Bab91, McM92, Pel94, Val91, BCCZ99, HNSY94, BCG95]. We can broadly classify four different methods to alleviate the state space:

- **Symbolic representations**
- **On the fly model checking**
- **Compositional reasoning**
- **Reduction methods**

In general, symbolic methods avoid the state space explosion problem by not explicitly representing the states of the model. Micmillan in his PhD thesis proposed to use symbolic representation for the state transition graph [McM92] . He showed that by using symbolic representations such as BDD much larger systems can be verified. Later by using original CTL model checking algorithms and their refinements, it was showed to verify examples with more than 10^{120} states.

On the fly model checking avoids the state space explosion problem by building only the necessary part of the system. In compositional reasoning, the objective is to verify local properties of the components of a system and then deduce global properties from these local properties. In principle, if we can deduce that local properties of components of a system are satisfied then the conjunction of local properties implies the satisfaction of global properties. One difficulty in this approach is that it is not often the case that local properties are not preserved at the global level [BCC98]. Reduction methods are well suitable for asynchronous systems. Reduction methods are used to transform the verification problem into an equivalent problem in a smaller state space.

In this thesis work, we use a structural reduction technique i.e., *slicing*. It helps to reduce a model syntactically with respect to the property. The resultant sliced model is used to verify given properties instead of complete model. Considering as a mean to alleviate state space explosion problem for model checking the sliced model has to preserve temporal properties and decrease its state space. To be specific with the objective of this work, we are interested to improve the model checking of APN models by means of Petri nets Slicing (PN slicing). Let us study *slicing* in general and then specifically PN slicing.

3.1 Overview and Background of Slicing

The term slicing was coined by M.Weiser for the first time in the context of program debugging [Wei81] and later refined by many authors [KL88, BG96, Bin98] . 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 [Wei81], let us consider an example program shown in Fig.3.2, 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 [Tip95, XQZ+05]).

```
(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).

Figure 3.2: Example program and its slice

3.2 Petri nets Slicing

Petri nets Slicing (PN slicing) is a syntactic technique used to reduce a PN model based on a given *criteria*. Informally, *criteria* (also termed as *slicing criterion*) is a property for which PN model is analyzed. A sliced part constitutes only that part of a PN model that may affect the *criteria*. Considering a property over PN model, we are interested to define a syntactically smaller PN model 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. Let us study a basic PN slicing algorithm presented in [Rak08].

The basic slicing algorithm starts with a PN model and a slicing criterion $Q \subseteq P$.

Algorithm 1: Basic slicing algorithm

BasicSlicing($\langle P, T, f, w, m_0 \rangle, Q$){
$T' \leftarrow \emptyset$;
$P' \leftarrow Q$;
$P_{done} \leftarrow \emptyset$;
while $((\exists p \in (P' \setminus P_{done}))$ **do**
  while $(\exists t \in ((^\bullet p \cup p^\bullet) \setminus T')$ **do**
    $P' \leftarrow P' \cup \{^\bullet t\}$;
    $T' \leftarrow T' \cup \{t\}$;
  end
  $P_{done} \leftarrow P_{done} \cup \{p\}$;
end
return $\langle P', T', f_{|P',T'}, w_{|P',T'}, m_{0_{|P'}} \rangle$;
}

In the basic slicing algorithm, initially T' (representing transitions set of the slice) is empty while P' (representing places set of the slice) contains the places extracted from the temporal formula of the property to be verified. The algorithm iteratively adds other *preset* transitions together with their *preset* places in T' and P'.

Let us take a simple example of Petri net model to explain the basic idea of PN slicing.

As shown in the Fig.3.3, 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.

3.2.1 Types of Slicing

Roughly, we can divide PN slicing in to two major classes (as shown in Fig.3.4), which are:

- **Static Slicing**
- **Dynamic Slicing**

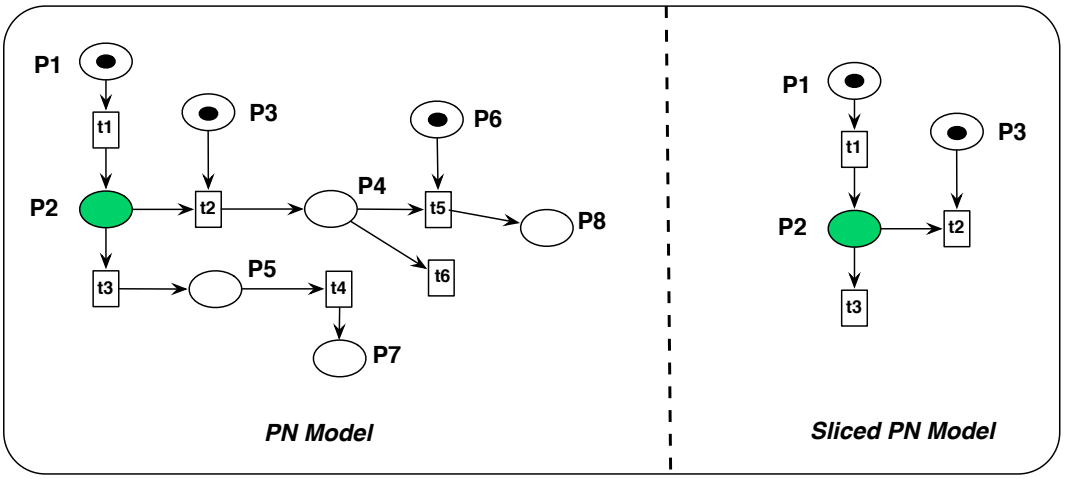


Figure 3.3: Example Petri net and its sliced model after applying basic slicing algorithm (Alg.1).

Static Slicing:

A slice is said to be static if the initial markings of places are not considered for generating the slice. Only a 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 sets 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.3.3 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.

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 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 produce

to change the marking of a criterion place in any execution starting from the initial marking.

Dynamic slicing can be useful 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 until the criterion places, a slice is generated. Backward slicing starts from the criterion places and then by backward traversal all the incoming transitions together with their input places, slice is obtained.

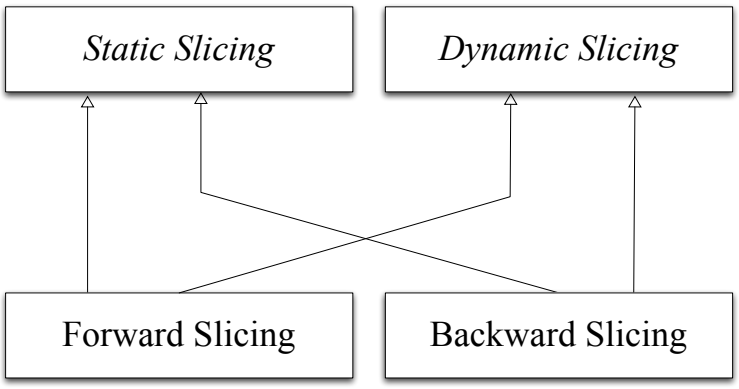


Figure 3.4: Types of PN slice

3.3 Petri nets Slicing Algorithms

In this section, we study existing algorithms for Petri nets Slicing (PN slicing) [CR94, LKCK00, Rak08, Rak11, Rak12, WCZX13, LOS+08]. The objective of every algorithm is to improve the verification process either by reducing or by partitioning a Petri nets (PN) model. These algorithms are divided into two groups with respect to the construction methodolody as described in the previous section (shown in Fig. 3.5).

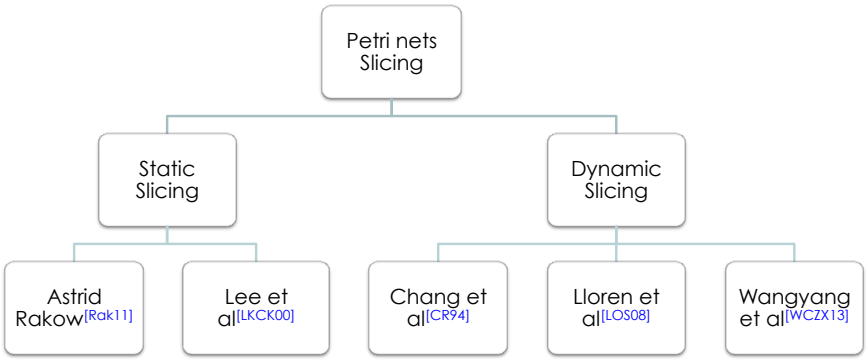


Figure 3.5: Slicing algorithms

3.3.1 Chang et al Slicing

Chang et al presented an algorithm for the first time for slicing Petri nets in the context of testing [CR94]. 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 2: Chang Slicing

Input: $ProcessPN[1], ..., ProcessPN[N]$.
$CS = \{t | t \in T, t \text{ is a communication transition}\}$;
Output: $S[1], S[2], \ldots, S[I]$.
Variables: $Mar = \{t | t \in T, t \text{ has a mark}\}$;
$TM = \{t | t \in T, t \text{ has a temporary mark}\}$;
$WS = \{t | t \in CS, t \text{ is in current process being scanned}\}$;
$Algorithm\_Slicing(ProcessPN[1], \ldots, ProcessPN[N], CS :$
$in; S[1], S[2], \ldots, S[I] : out)$;
$for(j \leftarrow 1 \text{ to } N)$ 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 $(CS \neq \emptyset \text{ and } terminate = false)$ **do**
  $I < -I + 1; S[I] \leftarrow \emptyset$;
  $Mar \leftarrow \emptyset; TM \leftarrow \emptyset$;
  $WS \leftarrow \emptyset; WS \leftarrow WS \cup \{t\}$;
  $Procedure\_findpath(ProcessPN[1], WS : in; PA : out)$;
  $S[I] \leftarrow S[1] \cup PA$;
  $M \leftarrow M \cup \{t | t \in ProcessPN[x], x = 1 \text{ and } t \text{ has relation with } PA\}$;
  $ProcedureScanning(ProcessPN[1], \ldots, ProcessPN[N] :$
  $in; CS, Mar, TM, S[I] : in \ \& \ out)$;
  $CS \leftarrow CS - CS \cap S[I]$;
end
$endslicing$;

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 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 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 the transitions having relations with the succeeding processes are marked. 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 con-

currency 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 [CR94]. Remark that the presented algorithm is linear time complex and the time complexity is $O((n/N))^N$.

Let us take a simple PN model (shown in Fig.3.6) and apply slicing algorithm 2 on it. The path shown with red doted line is put into the concurrency set.

3.3.2 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 [LKCK00]. The proposed approach may enable verification of properties such as boundedness and liveness, which may fail on unsliced Petri nets due to a state space explosion problem.

Algorithm 3: Lee Slicing

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

The basic idea of the 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 ,

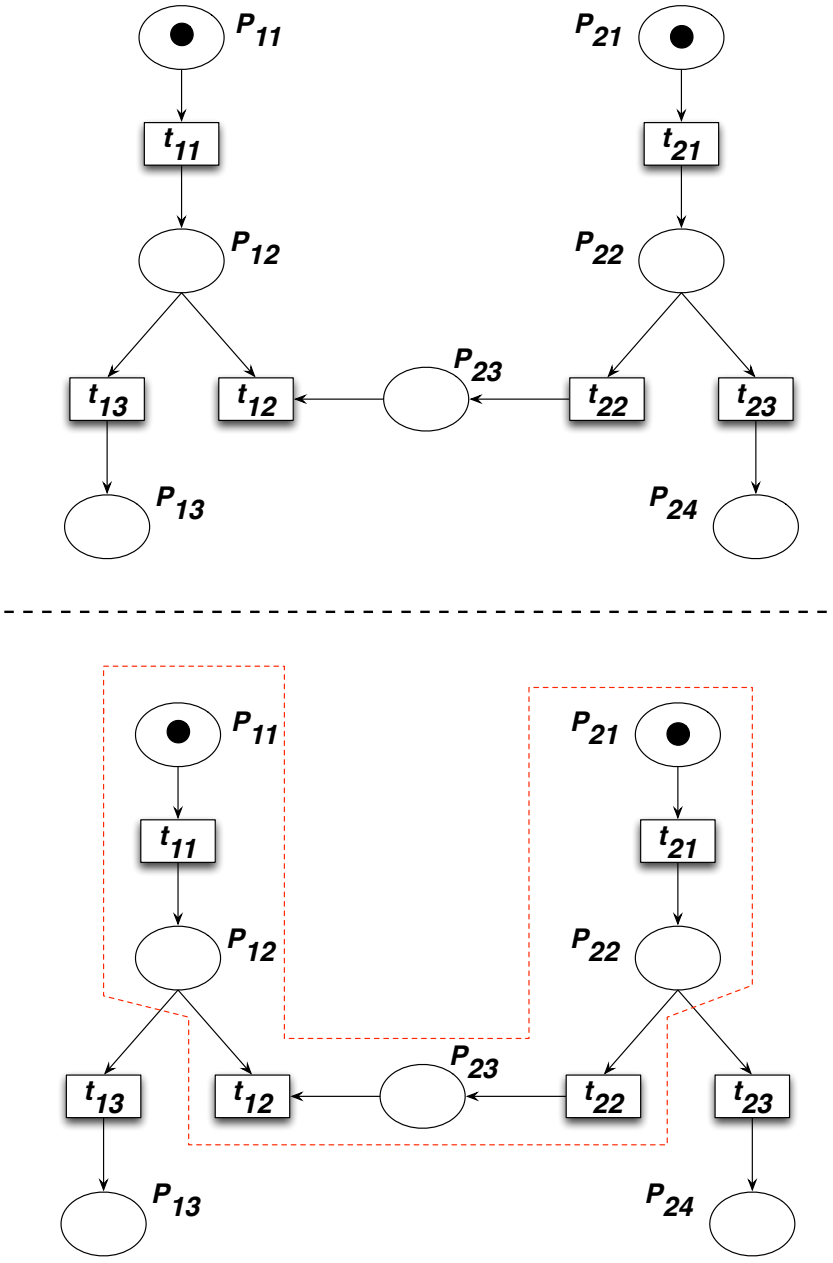


Figure 3.6: Example Petri net model and its concurrency set by applying Chang algorithm

for each uncovered place, it should be added into a slice to which it is connected by a minimal number of transitions (*by find_minimally_connected*). Like this, the slicing algorithm ensures that every place in *PN* belongs to some slices.

The proposed algorithm is linear time complex and the time complexity is $O(N)^3$. The time complexity depends on three procedures that are *find_minimal_invariant*, *find_smallest_invariant* and *find_minimally_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. However, in Modular Petri nets and Petri net slices they 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.

3.3.3 Llorens et al Slicing

Llorens et al presented a dynamic slicing algorithm for the first time in [LOS+08]. 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 account while in the second technique firing sequences are fixed to have a more reduced slice. The first technique includes three steps. In the first step, the basic algorithm given below computes a backward slice.

Algorithm 4: Bakward Slicing

GenerateSlice($\langle P, T, f, w, m_0\rangle, Crit$){
$T' \leftarrow \emptyset$;
$P' \leftarrow Crit$;
while ($\bullet P' \neq T'$) **do**
  $T' \leftarrow T' \cup \bullet P'$;
  $P' \leftarrow P' \cup \bullet T'$;
end
return $\langle P', T', f_{|_{P',T'}}, w_{|_{P',T'}}, m_{0|P'}\rangle$; }

Starting from the *criterion* place the algorithm iteratively include all the incoming transitions together with their input places until reaching a fix point. In the second

step a forward slicing is computed by the following algorithm.

Algorithm 5: Forward Slicing

GenerateSlice($\langle P, T, f, w, m_0 \rangle$){
$T' \leftarrow \{t \in T / m_0[t\rangle\}$;
$P' \leftarrow \{p \in P / m_0(p) > 0\} \cup T'\bullet$;
$T_{do} \leftarrow \{t \in T \setminus T' / \bullet t \subseteq P'\}$;
while ($T_{do} \neq \emptyset$) **do**
  $T' \leftarrow T' \cup T_{do}$;
  $P' \leftarrow P' \cup T\bullet_{do}$;
  $T_{do} \leftarrow \{t \in T \setminus T' / \bullet t \subseteq P'\}$
end
return $\langle P', T', f_{|_{P',T'}}, w_{|_{P',T'}}, m_{0|P'} \rangle$; }

Starting from the set of initially marked places the algorithm proceeds further by checking the enabled transitions. Then post sets 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 al fixes the firing sequences and can result in smaller slice. The algorithm is defined by auxiliary function and takes an initial marking together with the firing sequence denoted by σ and the set of places Q of the slicing criterion.

Algorithm 6: 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 backward 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 using case studies which is a big question mark on the usefulness and practicality.

We took an extremely simple PN model example and showed the resultant sliced PN models (by taking the intersection of sliced models obtained by applying forward and backward slicing algorithms). All the slices are generated considering place B as a

criterion place (see Fig.3.7).

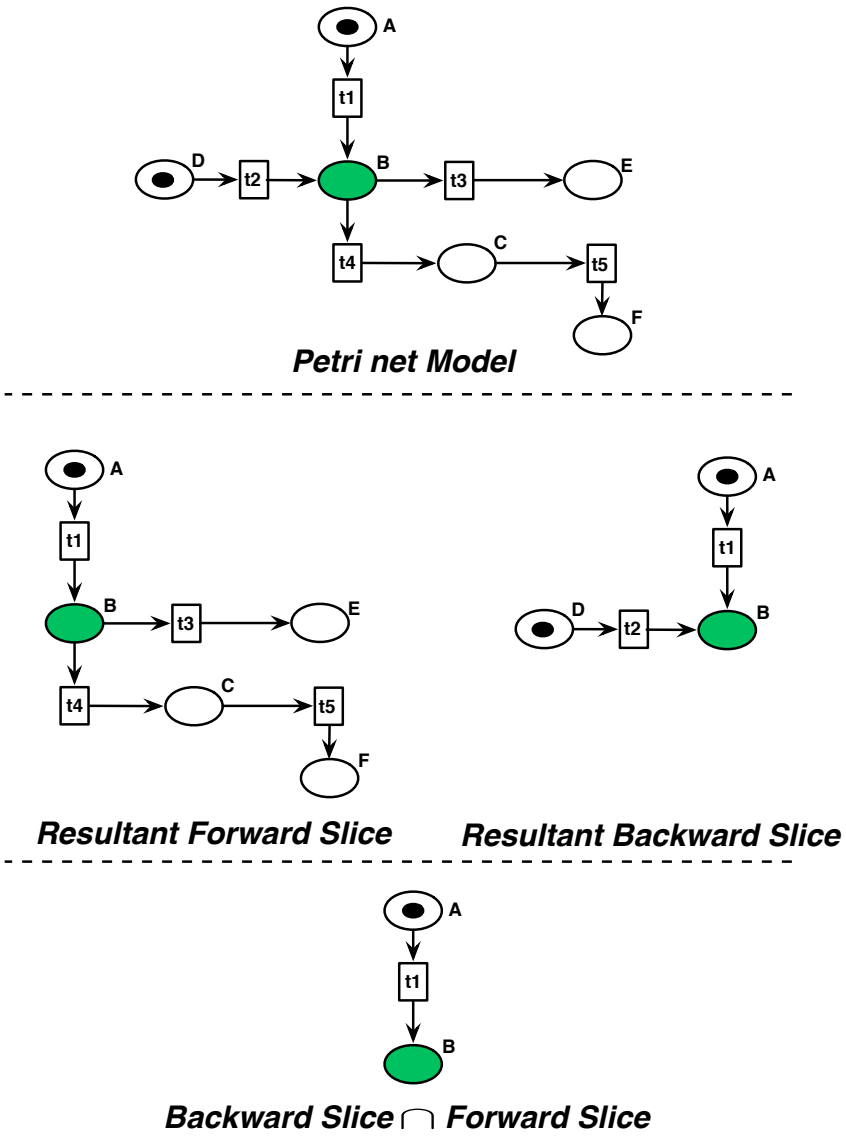


Figure 3.7: Example Petri net model and resultant sliced Petri net model by Llorens algorithm.

3.3.4 Rakow Slicing

A. Rakow presented two slicing algorithms, which are CTL$^*_{-X}$ slicing and safety slicing in [Rak12]. 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 a slicing criterion. A slicing criterion

consists of concerned places extracted from the temporal formula. A slice is generated by following dependencies backward from the criterion places.

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 marking of a place while *non-reading transitions* are transitions that change the marking of a place (as shown in Fig.3.8). Excluding the *reading transitions* and including the *non-reading transitions* during the slicing can certainly reduce the size of a slice. Formally, we can define *reading and non-reading transitions* such as:

Definition 3.3.1: (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 $\forall p \in (\bullet t \cup t\bullet), w(p,t) = w(t,p)$. Conversely, we call t a non-reading transition iff $\forall p \in (\bullet t \cup t\bullet), w(p,t) \neq w(t,p)$.

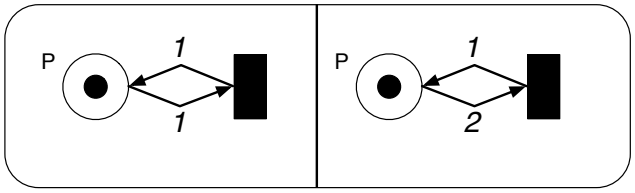


Figure 3.8: Reading and non-reading transitions of Petri net.

Algorithm 7: CTL*$_{-X}$ Slicing

```
GenerateSlice(⟨P, T, f, w, m0⟩, Crit){
T′, P_done ← ∅;
P′ ← Crit ;
while (∃p ∈ (P′ \ P_done)) do
    while (∃t ∈ (•p ∪ p•) \ T′) : w(p,t) ≠ w(t,p)) do
        P′ ← P′ ∪ •t;
        T′ ← T′ ∪ {t};
    end
    P_done ← P_done ∪ {p};
end
return ⟨P′, T′, f_|P′,T′, w_|P′,T′, m0|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 linear time com-

plex.

Algorithm 8: Safety Slicing

GenerateSlice($\langle P, T, f, w, m_0 \rangle, Crit$){
$T' \leftarrow \{t \in T \mid \exists p \in Crit : w(p,t) \neq w(t,p)\}$;
$P' \leftarrow \bullet T \cup Crit$;
$P_{done} \leftarrow Crit$;
while $(\exists p \in (P' \setminus P_{done}))$ **do**
  while $(\exists t \in (\bullet p \setminus T') : w(p,t) < w(t,p))$ **do**
    $P' \leftarrow P' \cup \bullet t$;
    $T' \leftarrow T' \cup \{t\}$;
  end
  $P_{done} \leftarrow P_{done} \cup \{p\}$;
end
return $\langle P', T', f_{|P',T'}, w_{|P',T'}, m_{0|P'} \rangle$; }

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 extremely simple example PN model and showed the resultant sliced PN models by applying basic, CTL*$_{-X}$ and safety slicing algorithms. All the slices are generated considering place B as a criterion place (see Fig.3.9). Clearly, safety slicing produces more smaller sliced PN model as compared to basic and CTL*$_{-X}$ sliced models but it preserves only safety properties.

3.3.5 Wangyang et al Slicing

Wangyang et al presented a backward dynamic slicing algorithm [WCZX13]. The basic idea of proposed algorithm is similar to the algorithm proposed by Lloren et al [LOS+08]. At first for both algorithms, a static backward slice 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 places in a Petri net model. After applying the static backward slicing algorithm, let us suppose that there are $n/2$ places. The algorithm of Llorens et al compute forward slice for n places whereas Wangyang

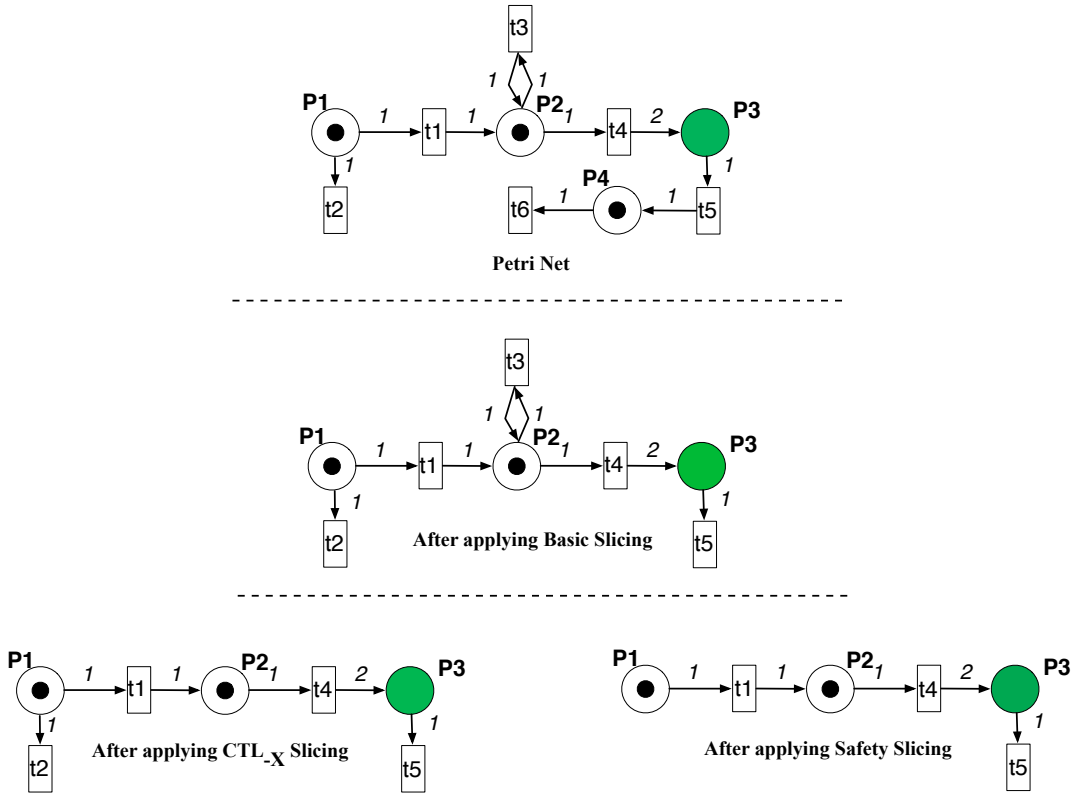


Figure 3.9: An example Petri net model and its sliced Petri net models by applying A.Rakow's proposed algorithms.

et al, algorithm will compute the forward slice only for $n/2$.

Algorithm 9: 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 initial marking m_0. Finally, backward dynamic slice can be obtained coupled with the initial marking and corresponding flow relation.

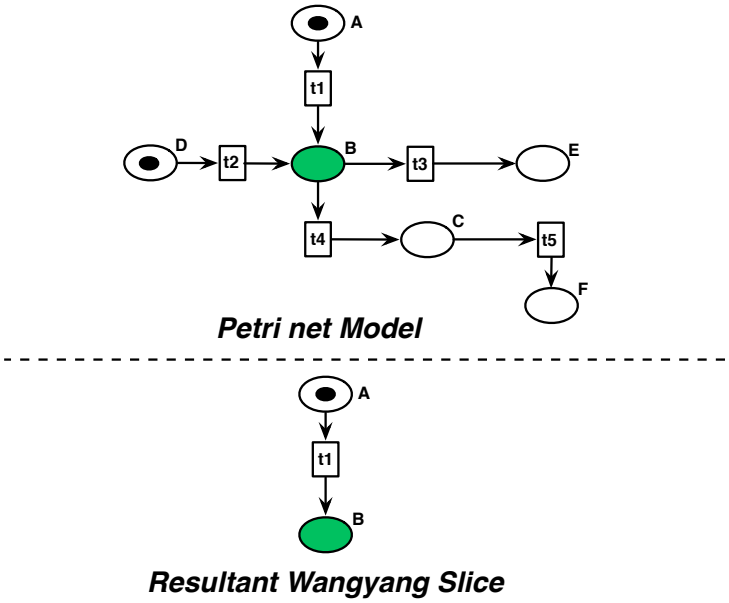


Figure 3.10: An example Petri net model and its sliced Petri net models by applying Wangyang's algorithm.

3.4 Comparison of Petri nets slicing algorithms

In this section, the static and dynamic slicing algorithms that were presented earlier are compared and classified. All the PN slicing constructions are proposed to improve testing or 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 to improve model checking extract a *slicing criterion* from the temporal description of properties. A *slicing criterion* consists of a set of places and then a slice is generated around them. We highlight in the Fig.3.11 different slicing algorithms that are designed for improving the model checking with respect to the slice size. We can notice that *Safety slicing* algorithm may generate the smallest slice as compared to other algorithms but preserve only safety properties. 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 to improve testing take directly the places or transitions as a *slicing criterion*.

Let us study the results summarized in the table.6.6. For each column, the table lists:

i) The names of proposed slicing algorithms that are presented earlier,

ii) The objective of every algorithm is presented i.e., to improve testing process or to alleviate the state space of model checking process,

iii) The reduction shows that either a model is reduced or there is no reduction by applying algorithm,

iv) Design context refers to the application of slicing algorithm with respect to Petri nets formalism; either it is designed for low-level or high-level Petri nets,

v) The properties that are preserved by the slicing construction are given. 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,

vi) The slicing type refers to the construction methodology i.e., either it is static or dynamic (as discussed earlier in the chapter) for slicing types) and is following backward propagation or forward (or both),

vii) The time complexity for each construction is presented. In general, every slicing algorithm proposed so far is polynomial time complex.

viii) The last column represents the existence of implementation for proposed algorithm.

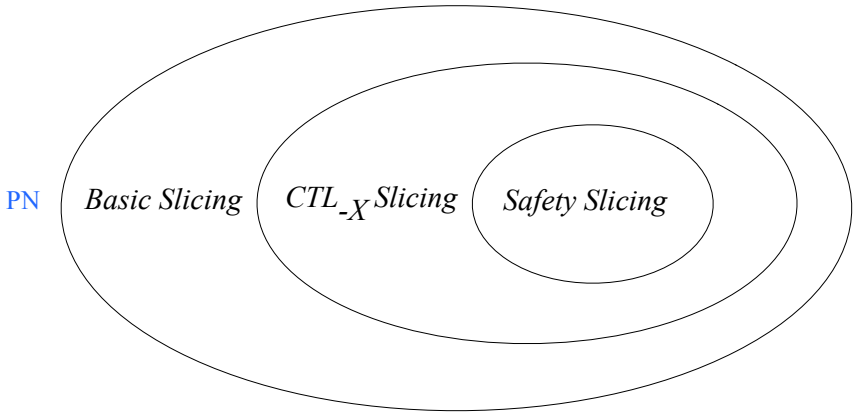


Figure 3.11: Petri net slicing algorithms w.r.t slice size

Agorithm	*Objective*	*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*
Llorens et al 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 3.1: Comparison of PN slicing Algorithms

Chapter 4

Property Based Model checking of Algebraic Petri nets

The beginning of wisdom is to desire it.

— Ibn Gabirol

Model checking is a convenient approach to analyze systems modeled in Algebraic Petri Net (APN). In general, APN model checking consists in a complete state space generation of an APN model to verify a given property. A typical drawback of model checking is its limit with respect to the state space explosion problem. As systems get moderately complex, completely enumerating their states states demands a growing amount of resources that in some cases makes model checking impractical both in terms of time and memory consumption.

The main advantage of model checking over traditional analysis approaches is that it is fully automatic. A lot of model checkers has been designed to analyze Petri net models [BHMR10, RWL+03, Mä02, BCC07]. In principle, every model checker explores all the possible states of a model to verify a given property and suffers with the state space explosion problem. An intense field of research is targeting to find ways to improve the performance of model checkers by reducing the state space. Recently, 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 Binary Decision Diagram (BDD) and has been successfully applied to Petri nets (PNs).

In this work, we propose a solution to improve the model checking by generating a partial state space which is sufficient to verify a property. The proposed approach is based on slicing an APN model by taking properties into consideration. The main advantage of our approach is that it can be considered as a pre-processing step towards model checking and can be easily added to any existing model checkers. Informally, a syntactically reduced APN model is obtained by applying a slicing algorithm. The

slicing algorithm takes APN model and temporal description of properties. The resultant sliced net is used to generate the state space to verify the property for which it is sliced. It is important to note that the resultant sliced net preserves certain class of properties such as *safety, liveness* or both, depending on the construction used by slicing algorithms. For example, some algorithms preserves *safety properties* only but generate smaller sliced net as compared to others.

As discussed in the chapter 3, slicing algorithms can be divided into two major classes, which are *static and dynamic slicing algorithms*. The static slicing algorithms are

Table 4.1: Proposed Slicing Algorithms for Algebraic Petri nets

Algorithm	*Preserved Prop*	*Type Slicing*
APN Slicing	LTL_{-X}	*Static*
Abstract Slicing	CTL^*_{-X}	*Static*
Safety Slicing	*Safety*	*Static*
Liveness Slicing	*Livenss*	*Static*
Concerned Slicing	*Particular*	*Dynamic*
Smart Slicing	*Particular*	*Dynamic*

designed to improve model checking whereas the dynamic slicing algorithms are designed to facilitate debugging. We propose several static and dynamic slicing algorithms for APNs as given in Tab: 4.1. In the first column of table, the name of proposed slicing algorithm is given, whereas in the second column a class of properties that are preserved by the algorithm are given. Consider for instance, the *APNSlicing* slicing algorithm preserves all the *safety* and *liveness* properties that can be represented by an LTL_{-X} formulas. The third column indicates the type of slicing either static or dynamic. The proposed static slicing algorithms can alleviate state space even for strongly connected nets and are proved not to increase the state space. It is important to note that the proposed algorithms can be applied to all classes of PNs with slight modifications.

Fig.4.1, gives an overview of the proposed approach for property based model checking of Algebraic Petri nets using a Process Flowchart. At first, an APN model is unfolded. The reason to unfold an Algebraic Petri net is to determine the ground substitutions of algebraic terms over the arcs to generate a smaller slice. Secondly, by taking properties into account, *criterion places* are extracted. Afterwards, slicing is performed for the *criterion places*. Subsequently, verification is performed on the sliced unfolded APN. The user may use the counterexample to refine the APN model

to correct the model to satisfy the property. Let us study in details the underlying

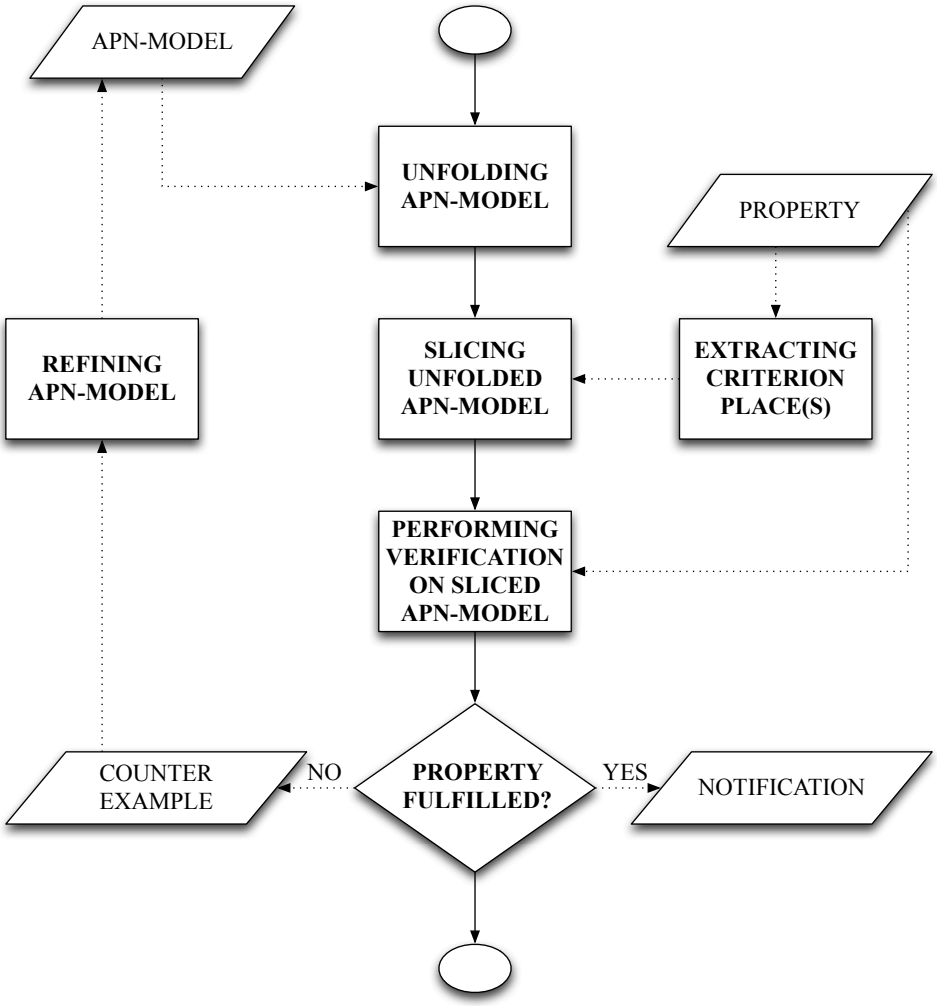


Figure 4.1: Process Flowchart of Property based model checking of Algebraic Petri nets

theory and techniques for each activity of the process.

4.1 Slicing Algebraic Petri nets

The main challenge in Petri nets Slicing (PN slicing) is to reduce the size of a resultant sliced net by preserving *safety* and *liveness* properties. Various algorithms are designed for slicing low-level Petri nets as discussed in the chapter 3. The main objective of existing algorithms is to reduce the size of resultant sliced net such that the state space required to verify a given property can be reduced. Rakow made the first improvement to reduce a slice size in the context of low-level Petri nets by introducing a notion of *reading and non-reading transitions*[Rak12]. The idea is to remove those transitions that consume and produce the same token to a place (i.e., reading transitions see definition 3.3.1) in the sliced net. The reading transitions are determined with the help of weights attached to the arcs of transitions.

One characteristic of APNs that makes them complex to slice as compared to low-level Petri nets is the use of multisets of algebraic terms over the arcs. In principle, algebraic terms may contain variables. Even though, we want to reach a syntactically reduced net (to be semantically valid) by identifying reading transitions, its reduction by slicing, needs to determine the possible ground substitutions of these algebraic terms. As shown in Fig.4.2 , both transitions are reading even if they are syntactically different.

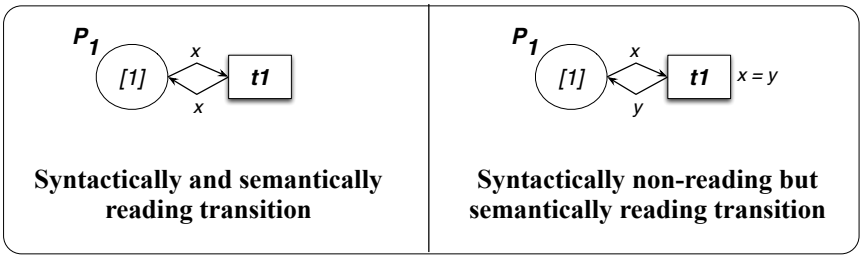


Figure 4.2: Syntactically and semantically reading transitions of Algebraic Petri nets

4.1.1 Partial Unfolding Algebraic Petri nets

The first step in property based model checking of Algebraic Petri nets is unfolding. Unfolding generates all possible firing sequences from the initial marking of the APNs, though maintaining a partial order of events based on the causal relation induced by the net, concurrency is preserved. In this work, we use partial unfolding approach used in AlPiNA (a symbolic model checker for Algebraic Petri nets) [BHMR10]. The AlPiNA allows a user to define partial algebraic unfolding and presumed bounds for infinite domains, using some aggressive strategies for reducing the size of large data domains.

Due to partial unfolding, there could be some domains that are not unfolded. For some cases, we are still able to identify *non-reading transitions* even if the domains are not

unfolded. If for example, we have a case where the multiplicities or cardinalities of terms in $\lambda(p,t), \lambda(t,p)$are different then we can immediately state $\lambda(p,t) \neq \lambda(t,p)$. But for some cases, we don't have such a clear indication of the inequality between $\lambda(p,t)$ and $\lambda(t,p)$, for example, in Fig.4.3, we see that $\lambda(p,t) = 1 + y$ and $\lambda(t,p) = 2 + x$ (defined over naturals). Both terms have the same multiplicity and cardinality, so we need to know for which values of the variables it would be a *non-reading transition*. In general, the evaluation of terms to check their equality for all the values is undecidable. For this particular case, we would like to have a set of constraints from a user. Informally, a constraints set denoted by CS, is a set of propositional formulas,

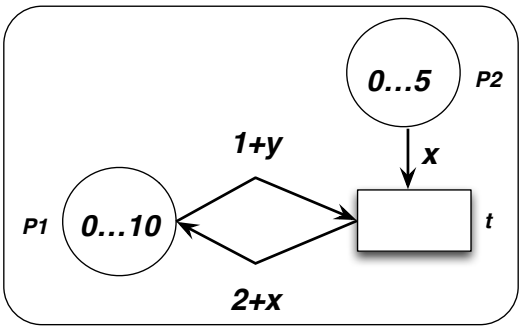


Figure 4.3: An example APN model with non-unfolded terms over the arcs

predicate formulas or any other logical formulas for certain specific values of variable assignments, describing the conditions under which we can evaluate terms to be equal or not. Consequently, the constraints set CS will help to identify under which cases the transitions can be treated as *non-reading*.

A function $eval : T_{OP,s}(X) \times T_{OP,s}(X) \times CS \rightarrow Bool$ is used to evaluate the equivalence of terms based on the constraints set. Let us take the same terms shown over the arcs in Fig.4.3, $term_1 = 1 + y$, $term_2 = 2 + x$ and a constraint set $CS = \{\exists y, x \in (0,\ldots,2) | y = x + 1\}$. It is important to note that we are not unfolding the domain but evaluating the terms for some specific values provided by user to identify *reading and non-reading transitions*. Of course, the user can provide sparse values too. Let us evaluate the terms $term_1$ and $term_2$ based on the constraints set CS provided. For all those values of x, y for which we get *eval* function result true are considered to be *reading transitions* and rest of them are *non-reading transitions*. It is also important to note that we include this step during the unfolding. The resulting unfolded *APN* will contain only *non-reading transitions* for the unfolded domains as shown in Fig.4.4. All the algorithms proposed in this thesis assume that such an unfolding takes place before the slicing. Since this is a step that is involved in the model checking activity anyway, we do not consider this assumption to be adding to the comparative complexity of the algorithm. In this section, we will make an extremely simple example of how the slicing algorithm works, starting from an APN and unfolding it.

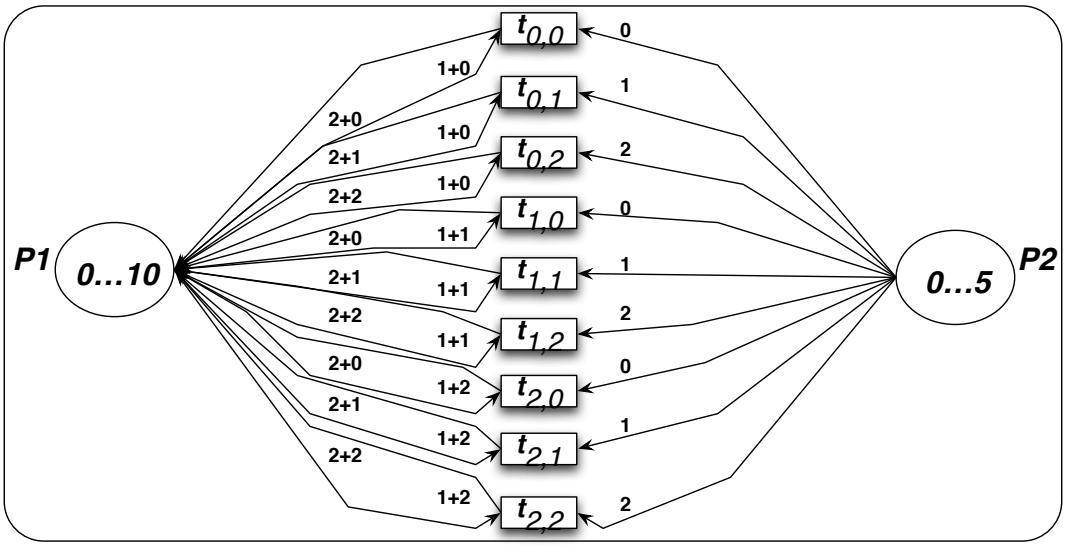


Figure 4.4: Resulting unfolded APN after applying the *eval* function

4.1.2 Example: Partially Unfolding an Algebraic Petri net

Fig. 4.5 shows an APN model. All places and all 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

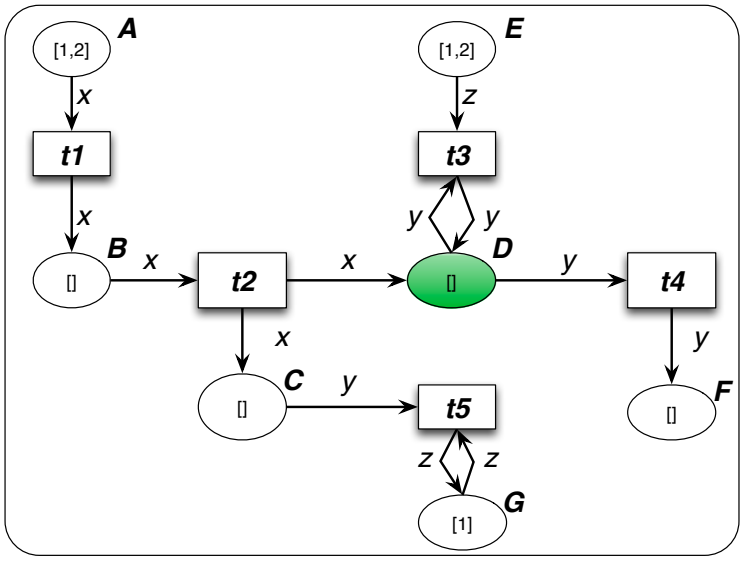


Figure 4.5: An example APN model (*APNexample*)

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 the 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 [BHMR10], 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 [BHMR10] 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 that satisfy the transition guards are kept. Third, the computed bindings are used to instantiate a binding-specific version of the transition. The resulting unfolded APN for this APNmodel is shown in Fig. 4.6. The interested reader can find details about the partial unfolding in [BHMR10].

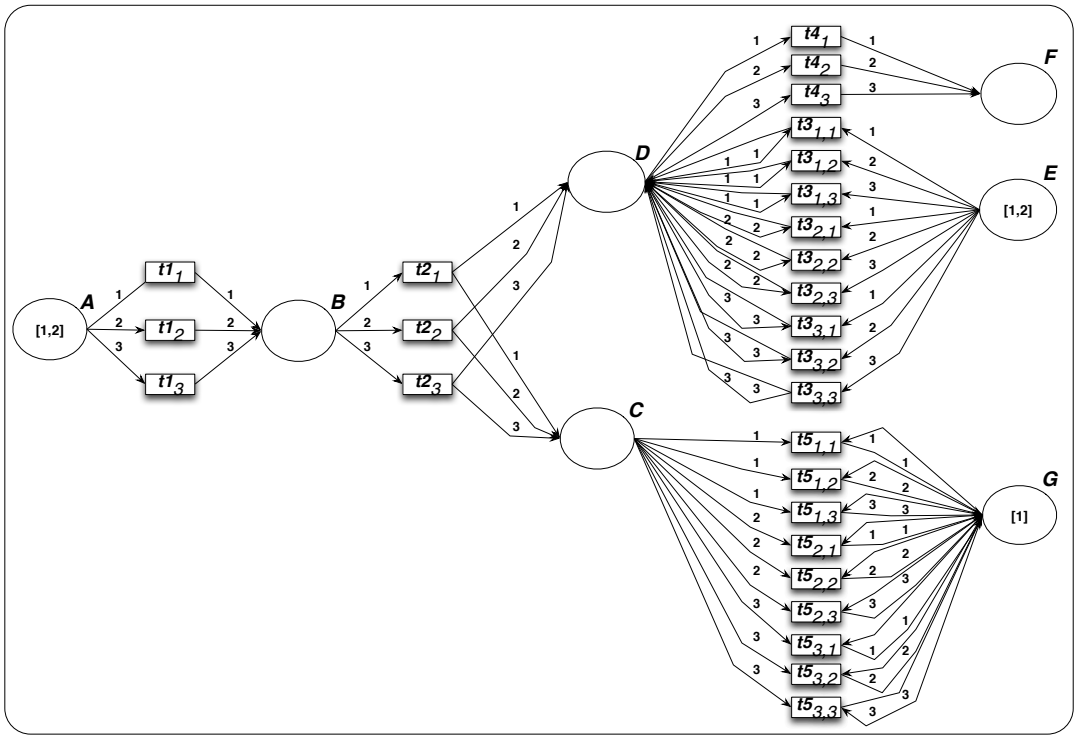


Figure 4.6: Partially unfolded example APN model (*UnfoldedAPN*)

4.2 Extraction of Criterion Places

In general, Petri net slicing aims to syntactically reduce a Petri net model based on a given criteria. A criteria can be a set of transitions or places (or both) that are provided by a user or it can be extracted automatically from a temporal description of properties. The resultant sliced Petri net model constitutes only that part of a model that may affect the criteria.

Let φ be a set representing CTL/LTL formulas and P represents set of places. A function $extract : \varphi \rightarrow P$ is used to extract places from an CTL/LTL formula. Let us take an example CTL formula and see extract places from it by using *extract* function. Consider for an example, $\phi = \mathbf{G}(P2 \neq \emptyset)$. By using *extract* function a criteria can be extracted such as $extract(\phi) = \{P2\}$. In this particular formula the only criterion place is $P2$ and a slicing algorithm will start from this place.

4.3 Static Slicing on Partially Unfolded Algebraic Petri nets

The basic idea of slicing algorithm is to start by identifying which places in the unfolded APN model are directly concerned by a property. These places constitute the *slicing criterion*. The algorithm will then take 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 (Note: see basic algorithm in section 3).

We refine the slicing construction by distinguishing between *reading* and *non-reading transitions*. The conception of *reading and non-reading transitions* is some what similar notion introduced in [Rak12]. The main difference is that we adapt the notion of *reading and non-reading transitions* in the context of APNs. Informally, *reading transitions* are not supposed to change the marking of a place. On the other hand *non-reading transitions* are subject to change the markings of a place. Unfolding of APNs helps us to identify syntactically *reading and non-reading transitions*. In our proposed slicing construction, we discard *reading transitions* and include only *non-reading transitions* because they are affecting the property satisfaction.

4.3.1 The slicing algorithm: APNSlicing

The slicing algorithm starts with an unfolded APN and a slicing criterion $Q \subseteq P$.

Let $Q \subseteq P$ a non empty set called slicing criterion. We can build a slice for an *unfolded*

APN based on Q, using following algorithm:

Algorithm 10: APN slicing algorithm

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 $\langle SPEC, P', T', F_{|_{P',T'}}, asg_{|_{P'}}, cond_{|_{T'}}, \lambda_{|_{P',T'}}, m_{0_{|_{P'}}}\rangle$;
}

Initially, T' (representing transitions set of the slice) contains set of all *pre and post* transitions of the given criterion place. Only *non-reading* transitions are added to T' set. P'(representing places set of the slice) contains all *preset* places of transitions in T'. The algorithm then iteratively adds other *preset* transitions together with their *preset* places in T' and P'. Remark that the *APNSlicing* algorithm has linear time complexity.

Considering the APN-Model shown in fig. 4.5, let us now take two example properties and apply our proposed algorithm on them. 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 them such as:

$\phi_1 = \mathbf{G}(\forall token \in m(D)/token < 5)$.

$\phi_2 = \mathbf{F}(|m(D)| \neq \emptyset)$

For both properties, the slicing criterion $Q = \{D\}$, as D is the only place concerned by the properties. Therefore, the application of *APNSlicing(UnfoldedAPN, D)* returns *SlicedUnfoldedAPN* (shown in Fig. 4.7), which is smaller than the original *UnfoldedAPN* shown in Fig. 4.6).

Transitions $t3_{1,1}, t3_{1,2}, t3_{1,3}, t3_{1,3}, t3_{2,1}, t3_{2,2}, t3_{2,3}, t3_{3,1}, t3_{3,2}, t3_{3,3}, t5_{1,1}, t5_{1,2}$, $t5_{1,3}, t5_{2,1}, t5_{2,2}, t5_{2,3}, t5_{3,1}, t5_{3,2}, t5_{3,3}$, and places C, E, F, G has been sliced away. The proposed algorithm determines a slice for any given criterion $Q \subseteq P$ and always terminates. It is important to note that the reduction of net size depends on the structure of the net and on the size and position of the slicing criterion within the net.

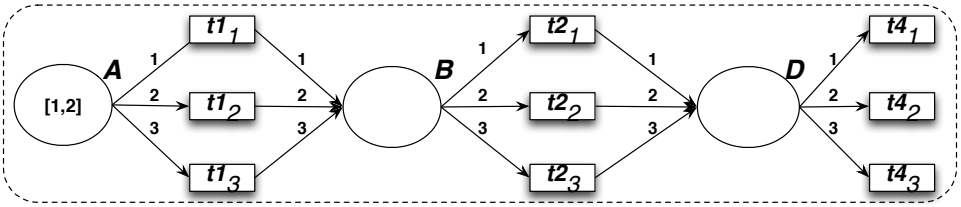


Figure 4.7: Resultant Sliced Unfolded example APN model (*SlicedUnfoldedAPN*)

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.6.6, 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.

Table 4.2: Comparison of number of states required to verify the property with and without APNslicing.

Properties	*No of states required without slicing*	*No of states required with slicing*
ϕ_1	*148*	16
ϕ_2	*148*	16

4.3.2 Proof of the preservation of properties by APNslicing algorithm

To allow the verification by slice, we have to make restrictions on the formulas and on admissible firing sequences in terms of fairness assumptions. The original Algebraic Petri net has more behaviors than the sliced net, as we intentionally do not capture all the behaviors. The proof of preservation of properties is similar to the proof done by Rakow in [Rak11] but we adapt in the context of unfolded APNs.

Definition 4.3.1:
Let A be the set of atomic propositions. Let ϕ, ϕ_1, ϕ_2 be CTL formulas. The function scope associates with a CTL formula ϕ, the set of atomic propositions used in ϕ i.e. $scope : CTL \rightarrow \mathcal{P}(A)$.

$scope(a) = \{a\}$;

$scope(\otimes\phi) = scope(\phi)$ with $\otimes \in \{\neg, X\}$;

$scope(\phi_1 \otimes \phi_2) = scope(\phi_1) \cup scope(\phi_2)$ with $\otimes \in \{\wedge, U\}$.

Definition 4.3.2:
Let apn be a marked unfolded APN. Let apn' be its sliced net for a slicing criterion $Q \subseteq P$. Let $\sigma = t_1 t_2 t_3 \ldots$be a firing sequence of apn and m_i the markings with $m_i[t_{i+1}\rangle m_{i+1}, \forall i, 0 \leq i < |\sigma|$. σ is slice-fair w.r.t apn' iff

$$\begin{cases} either\ \sigma\ is\ finite\ and\ m_{|\sigma|}\ does\ not\ enable\ any\ transition t \in T'; \\ or\ \sigma\ is\ infinite\ and\ if\ it\ permanently\ enables\ some\ t \in T', \\ \ it\ then\ fires\ infinitely\ often\ some\ transition \\ of\ T'\ (which\ may\ or\ may\ not\ be\ the\ same\ as\ t). \end{cases}$$

Slice-fairness is a very weak fairness notion. Weak fairness determines that every transition $t \in T$ of a system, if permanently enabled, has to be fired infinitely often, slice-fairness concerns only the transitions of the slice, not of the entire system net and if a transition $t \in T$ of the slice is permanently enabled, some transitions of the slice are required to fire infinitely often but not necessarily t.

Definition 4.3.3:
Let apn be a marked unfolded APN and ϕ a CTL formula. Let σ be a firing sequence of apn. $apn \models_{sf} \phi$ (slice-fairly) $\Leftrightarrow \sigma \models \phi$ for every slice-fair firing sequence (not necessarily maximal) .

Definition 4.3.4:
Let apn and apn' be two marked unfolded APNs with $T' \subseteq T$ and $P' \subseteq P$. Let $TM : (T^* \cup T^w) \rightarrow (T'^* \cup T'^w)$ such that a finite or infinite sequence of transitions σ

is mapped onto the transition sequence σ' with σ' being derived from σ by omitting every transition $t \in T \setminus T'$ and $PM : \mathbb{N}^{|P|} \rightarrow \mathbb{N}^{|P'|}$ such that a marking m of apn is projected onto the marking m' of apn' with $m' = m \mid_{p'}$. We define the function: $\text{slice}_{(apn,apn')} \in \{TM \cup PM\}$.

The function *slice* is used to project markings and firing sequences of a net apn onto the markings and firing sequences of its slices.

Proposition 4.3.1:
Let apn be a marked unfolded APN. Let apn' be its sliced net for a slicing criterion $Q \subseteq P$. Let σ be a weakly fair firing sequence of apn. σ is slice fair with respect to apn'.

Proof: Let us assume, σ is not slice-fair. In case σ is finite this means that $m_{|\sigma|}[t\rangle$ for a transition $t \in T'$. In case σ is infinite, there is permanently enabled transition $t \in T'$ but all transitions of T' are fired finitely often including t. So both cases contradict the assumption that σ is weakly fair.

Lemma 4.3.1:
Let apn be a marked unfolded APN and let apn' be its sliced net for a slicing criterion $Q \subseteq P$. The coefficients c_{ij} of the incidence matrix equal to zero for all places $p_i \in P'$ and transitions $t_j \in T \setminus T'$.

Proof: Let apn' be its sliced net for a slicing criterion $Q \subseteq P$. A transition $t \in T$ is also an element of $T' \subseteq T$, if it is a *non-reading* transition of a place $p \in P'$. Thus a transition $t \in T \setminus T'$ either is not connected to a place $p \in P'$ or it is a *reading* transition.

Lemma 4.3.2:
Let apn be a marked unfolded APN and let apn' be its sliced net for a slicing criterion $Q \subseteq P$. Let m be a marking of apn and m' be a marking of apn' with $m' = m \mid_{p'}$. $m[t\rangle \Leftrightarrow m'[t\rangle, \forall t \in T'$.

Proof: Let apn be its sliced net for a slicing criterion $Q \subseteq P$. Since a transition $t \in T'$ has the same preset places in apn and apn' by the slicing algorithm *APNSlicing*, $m' = m \mid_{p'}$ implies $m[t\rangle \iff m'[t\rangle$.

Every firing sequence σ of apn projected onto the transitions of T' is also a firing sequence of slice net apn'. The resulting markings m and m' assign the same number of tokens to places $p' \subseteq P$.

Proposition 4.3.2:
Let apn be a marked unfolded APN and let apn' be its sliced net for a slicing criterion $Q \subseteq P$. Let σ be a firing sequence of apn and let m be a marking of apn. $m_0[\sigma\rangle m \Rightarrow m_0 \mid_{p'} [slice(\sigma)\rangle m \mid_{p'}$.

Proof: We prove this Proposition by induction over the length l of σ. Let apn be a marked unfolded APN, σ be a firing sequence of apn.

$l = 0$: In this case slice(σ) equals ϵ. Thus the initial marking of apn and apn' is generated by firing ϵ. By defintion 4.3.4 and the slicing algorithm $APNSlicing$, $m'_0 = m_0 \mid_{p'}$.

$l \to l+1$: Let σ be a firing sequence of length l and m_l be a marking of apn with $m_0[\sigma\rangle m_l$. Let t_{l+1} be a transition in T and m_{l+1} a marking of N such that $m_l[t_{l+1}\rangle m_{l+1}$. By induction hypothesis, $m'_0[slice(\sigma)\rangle m'_k$ with $m_l \mid_{p'} = m'_k$. If t_{l+1} is an element of T', it follows by Lemma 4.3.2, that m'_k enables t_{l+1}, since m_l enables t_{l+1}. The resulting marking m'_{k+1} is determined by $m'_{k+1}(P'_i) = m'_k(P'_i) + c_{i\,l+1}, \forall p_i \in P'$ and m_{l+1} is determined by $m_{l+1}(i) = m_l(i) + c_{i\,l+1}, \forall p_i \in P'$.

Since $m_{l|_{P'}} = m'_k$, it thus follows that $m_{l_{l+1}|_{P'}} = m'_{k+1}$. If t_{l+1} is an element of $t \in T \setminus T'$, then it must be a reading transition for all $p \in P$; $slice(\sigma) = slice(\sigma t_{l+1})$ and thus $m'_0[slice(\sigma t_{l+1})\rangle m'_k$ a transition $t \in T \setminus T'$ can not change the marking of on any place $p \in P'$. By Lemma 4.3.1 and the resultant markings, $m_{l_{l+1}|_{P'}} = m'_l \mid_{P'}$. □

A firing sequence σ' of the slice net apn' is also a firing sequence of apn. The resulting markings of σ' on apn and apn', respectively assigns the same markings to places $p \in P'$.

Proposition 4.3.3:
Let apn be a marked unfolded APN and let apn' be its sliced net for a slicing criterion $Q \subseteq P$. Let σ' be a firing sequence of apn' and let m' be a marking of apn'.

$m'_0[\sigma'\rangle m' \Rightarrow \exists m \in \mathbb{N}^{|P|} : m' = m_{l|_{P'}} \wedge m_0[\sigma'\rangle m$.

Proof: We prove this Proposition by induction over the length l of σ'.

$l = 0$: The empty firing sequence generates the marking m_0 on apn and the marking m'_0, which is defined as $m_{0|_{P'}}$, on apn', by definition 4.3.4.

$l \to l+1$: Let $\sigma' = t_1 \ldots t_{l+1}$ be firing sequence of apn' with length $l+1$. Let m'_l and m'_{l+1} be markings of apn' such that $m'_0[t_1 \ldots t_l\rangle m'_l[t_{l+1}\rangle m'_{l+1}$. Let m_l be the marking of apn with $m_0[t_1 \ldots t_l\rangle m_l$ and $m_{l|_{P'}} = m'_l$, which exists according to the induction hypothesis. Lemma 4.3.2, m_l enables t_{l+1}. The marking m_{l+1} satisfies $m_{l+1}(P_i) = m_l(P_i) + c_{i\,l+1}, \forall p_i \in P'$ and m'_{l+1} satisfies $m'_{l+1}(P_i) = m'_l(P_i) + c_{i\,l+1}, \forall s_i \in P'$. With $m_{l|_{P'}} = m'_l$, it follows that $(m_{l+1} \mid_{P'})$ is equal to m'_{l+1}. □

Proposition 4.3.4:
Let apn be a marked unfolded APN and let ϕ be a CTL_x formula such that $scope(\phi) \subseteq P$. Let apn' be its sliced net for a slicing criterion $Q \subseteq P$ where $Q = scope(\phi)$. Let σ be a firing sequence of apn. Let us denote the sequence of markings by $\mathcal{M}(\sigma)$. Then, $\mathcal{M}(\sigma) \models \phi \Leftrightarrow \mathcal{M}(slice(\sigma)) \models \phi$.

Proof: We prove this Proposition by induction on the structure of ϕ. Let $\sigma = t_1 t_2 \ldots$ and $slice(\sigma)$ be $\sigma' = t'_1 t'_2 \ldots$. Let $\mathcal{M}(\sigma) = m_0 m_1 \ldots$ and $\mathcal{M}(\sigma') = m'_0 m'_1 \ldots$.

$\phi = true$: In this case nothing needs to be shown. $\phi = \neg\psi, \phi = \psi_1 \wedge \psi_2$: Since the satisfiability of ϕ depends on the initial marking of $scope(\phi)$ only and $scope(\phi) \subseteq P' \subseteq P$, both directions hold.

$\phi = \psi_1 U \psi_2$: We assume that $\mathcal{M}(\sigma') \models \psi_1 U \psi_2$. We can divide up σ' such that $\sigma' = \sigma'_1 \sigma'_2$ with $m'_{|\sigma'_1|} m'_{|\sigma'_1|+1} \ldots \models \psi_2$ and $\forall i, 0 \leq i < |\sigma'_1| : m'_i m'_{i+1} \ldots \models \psi_1$. There are transition sequences σ_1 and σ_2 such that $\sigma = \sigma_1 \sigma_2$, $slice(\sigma_1) = \sigma'_1$, $slice(\sigma_2) = \sigma'_2$ and σ_1 does not end with a transition $t \in T \setminus T'$.

By proposition 4.3.2, it follows that $m'_{|\sigma'_1|} = (m_{|\sigma_1|} \,|_{P'})$. Since $m'_{|\sigma'_1|} m'_{|\sigma'_1|+1} \ldots \models \psi_2, m_{|\sigma_1|} m_{|\sigma_1|+1} \ldots \models \psi_2$ by induction hypothesis. Let ϱ be a prefix of σ_1 such that $|\varrho| < |\sigma_1|$. Let ϱ' be $slice(\varrho)$. The firing sequence ϱ truncates at least one transition $t \in T'$, consequently $|\varrho'| < |\sigma'_1|$. Since $m'_{|\varrho'_1|} m'_{|\varrho'_1|+1} \ldots \models \psi_1, m_{|\varrho|} m_{|\varrho|+1} \ldots \models \psi_1$ by the induction hypothesis. Analogously, it can be shown that $\mathcal{M}(\sigma) \models \psi_1 U \psi_2$ implies $\mathcal{M}(\sigma') \models \psi_1 U \psi_2$. □

Proposition 4.3.5:
Let apn be a marked unfolded APN and let apn' be its sliced net for a slicing criterion $Q \subseteq P$. Let σ' be a maximal firing sequence of apn'. σ' is a slice-fair firing sequence of apn.

Proof: Let $\sigma' = t_1 t_2 \ldots$. Let m'_i be the marking of apn', such that $m'_i[t_{i+1}\rangle m'_{i+1}, \forall i, 0 \leq i < |\sigma'|$. By Proposition 4.3.3 σ' is a firing sequence of apn. Let m_i be the marking of apn, such that $m_i[t_{i+1}\rangle m_{i+1}, \forall i, 0 \leq i < |\sigma'|$. In case σ' is finite, $m'_{|\sigma'|}$ does not enable any transition $t' \in T'$.

By Lemma 4.3.2, $m_{|\sigma'|}$ does not enable any transition $T' \in T'$, If σ' is infinite it obviously fires infinitely often a transition $t' \in T'$ and thus is slice-fair. □

Proposition 4.3.6:
Let apn be a marked unfolded APN and let apn' be its sliced net for a slicing criterion $Q \subseteq P$. $Slice(\sigma)$ is maximal firing sequence of apn'.

Proof: Let $\sigma = t_1 t_2 \ldots$ with $m_i[t_{i+1}\rangle m_{i+1}, \forall i, 0 \leq i < |\sigma|$. By Proposition 4.3.2, $slice(\sigma)$ is a firing sequence of apn'. Let $slice(\sigma)$ be $\sigma' = t'_1 t'_2 \ldots$ with $m'_i[t'_{i+1}\rangle m'_{i+1}$,

$\forall i, 0 \leq i < |\sigma|$. Let us assume σ' is not a maximal firing sequence of apn'. Thus σ' is finite and there is a transition $t' \in T'$ with $m'_{|\sigma'|}[t'\rangle$. Let σ_1 be the smallest prefix of σ such that $slice(\sigma_1)$ equals σ'.

By Proposition 4.3.2 ($m_{|\sigma_1|} \mid_{P'} = m'_{|\sigma'|}$. By Lemma 4.3.2, and the state equation it follows, that ($m_{|\sigma_1|} \mid_{P'} = m'_{|\sigma'|+1} = \ldots$). So t' stays enabled for all markings m_j with $|\sigma_1| \leq j \leq |\sigma|$ but is fired finitely many times only. This is a contradiction to the assumption that σ is slice-fair. □

Theorem 4.3.1:
Let apn be a marked unfolded APN and let ϕ be a CTL formula such that $scope(\phi) \subseteq P$. Let apn' be its sliced net for a slicing criterion $Q \subseteq P$. Let ψ be a CTL_{-X} formula with $scope(\psi) \subseteq P$.

$apn \models_{sf} \phi \Rightarrow apn' \models_{sf} \phi$.

$apn \models_{sf} \psi \Leftarrow apn' \models_{sf} \psi$.

Proof: We first show " $apn \models_{sf} \phi \Rightarrow apn' \models_{sf} \phi$ ". Let us assume that $apn \models_{sf} \phi$ holds. Let σ' be a maximal firing sequence of apn'. Since σ' is a slice-fair firing sequence of apn by Proposition 4.3.5 $\mathcal{M}(\sigma') \models \phi$. Let us now assume $apn' \models_{sf} \psi$. Let σ be a slice-fair firing sequence of apn. By Proposition 4.3.6, $slice(\sigma)$ is maximal firing sequence of apn' and thus satisfies ψ. By Proposition 4.3.4, it follows that σ satisfies ψ. □

If we assume slice-fairness then verification is possible under interleaving semantics. A firing sequence σ is fair w.r.t T' , if σ is maximal and if σ eventually permanently enables a $t' \in T'$, a transition $t \in T'$ will be fired infinitely often (t may not equal t'). Unfolded APN $\models_{sf} \phi$ w.r.t. T' holds if all fair firings sequences of apn, more precisely, their corresponding traces satisfy ϕ. □

4.3.3 Abstract Slicing on Unfolded APNs

Abstract slicing has been defined as a *static slicing algorithm*. The objective is to improve the model checking of Algebraic Petri Nets (APNs) by developing a more refined slicing algorithm. 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 [Rak12], and later adapted by us in the context of APNs in [KR13]. Informally, *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.4.8). 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.

We introduce a new notion of *neutral transitions* to reduce the slice size. 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 transition 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 4.3.5:
(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., $t \in T \wedge \exists p \exists p' / p \in^\bullet t \wedge p' \in t^\bullet \wedge |p^\bullet| = 1 \wedge |{}^\bullet t| = 1 \wedge |t^\bullet| = 1 \wedge \lambda(t,p) = \lambda(t,p')$.

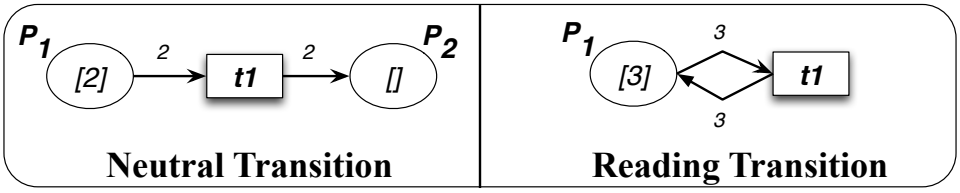


Figure 4.8: Neutral and Reading transitions of Unfolded APN

We extend the existing slicing algorithm by using *neutral transitions* and *reading transitions* (shown in Fig. 4.9). The advantage of using *neutral and reading transitions* together is smaller resultant slice. The more smaller slice means more reduction in the state space. Thus, abstract slicing is an improved version of slicing algorithms. Later, we show a comparison of abstract slicing results with the existing slicing results.

Figure 4.9: Abstract slicing construction methodology

4.3.4 The Slicing Algorithm: AbstractSlicing

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 11: 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 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 unfolded APN-Model shown in fig. 4.6, let us now take two example properties ϕ_1 and ϕ_2 (given in the previous section) and apply abstract slicing algorithm and compare the reduction in terms of state space. For both properties, the slicing criterion $Q = \{D\}$, as D is the only place concerned by the properties. Therefore, the application of *Abstractslice(UnfoldedAPN, D)* returns *SlicedUnfoldedAPN* (shown in Fig. 4.10*(A,B,C,D)*), which is smaller than the original *UnfoldedAPN* shown in Fig. 4.6). At first, all the reading transitions are removed that are attached to the criterion place. In this particular example, we remove transitions $t3_{1,1}, t3_{1,2}, t3_{1,3}, t3_{2,1}, t3_{2,2}, t3_{2,3}, t3_{3,1}, t3_{3,2}, t3_{3,3}$, and places F, E, the resultant net is shown in Fig. 4.10*(B)*). The next step is to iteratively include all the incoming transitions together with with their places. As a result, we remove $t5_{1,1}, t5_{1,2}, t5_{1,3}, t5_{2,1}, t5_{2,2}, t5_{2,3}$, $t5_{3,1}, t5_{3,2}, t5_{3,3}$, and place C as shown in Fig. 4.10*(C)*). Finally, neutral transitions are identified and their places are combined. In this example, transitions $t1_1, t1_2, t1_3$, are neutral transitions and their places are combined as shown in Fig. 4.10*(D)*).

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.6.7, 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.

Table 4.3: 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

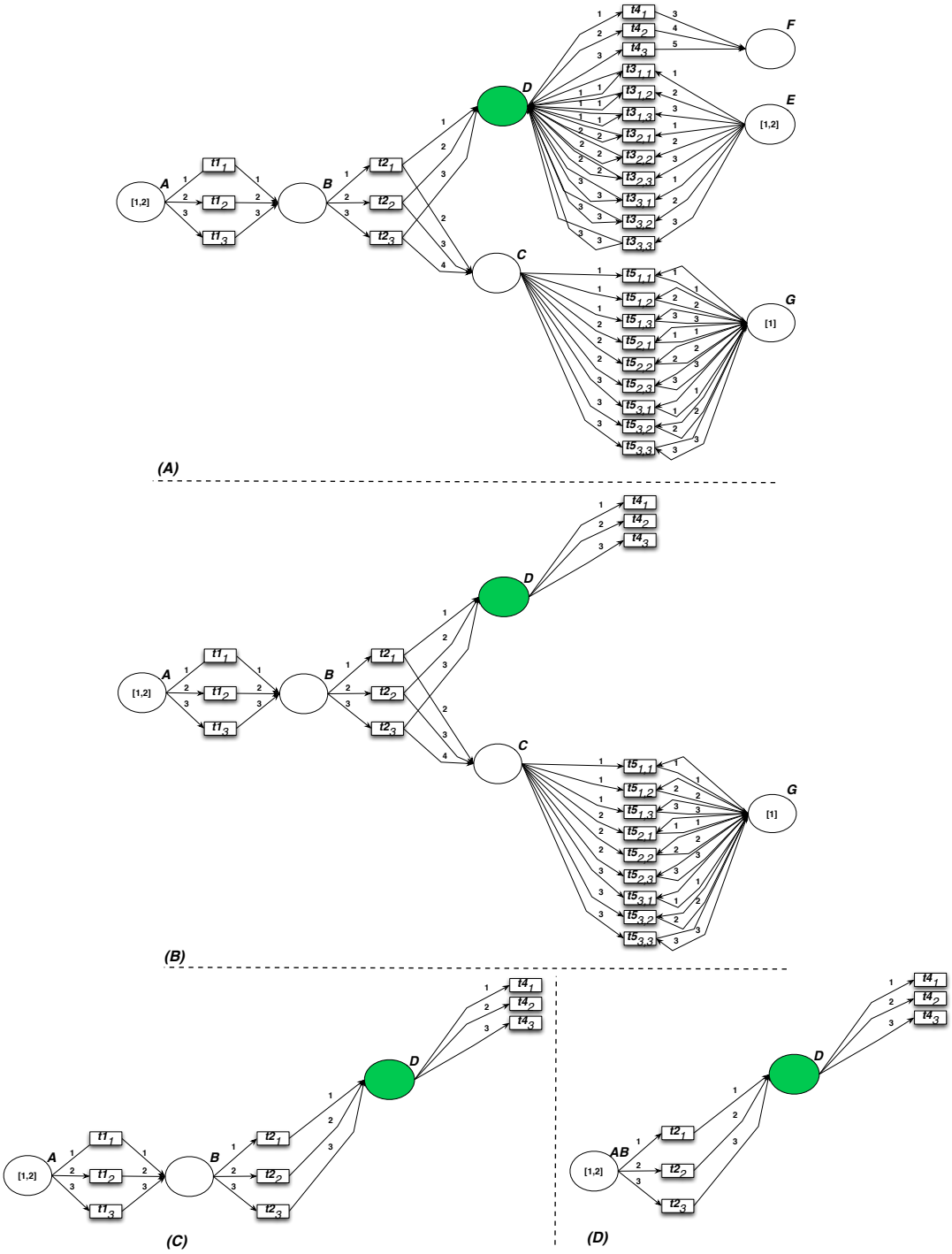


Figure 4.10: The sliced unfolded APNs (by applying *abstract slicing*)

4.3.5 Proof of the preservation of properties by abstractslicing algorithm

The abstract slicing algorithm is designed to generate a smaller slice net as compared to *APNSlicing* algorithm (given in the previous section). We can say that *Abstract slicing* algorithm is an advancement of *APNSlicing* algorithm. Notably, both algorithms preserves CTL_{-X} properties.

Theorem 4.3.2:
Let apn be a marked unfolded APN and Let $apn' = APNSlice(apn, Q)$ be its sliced net for a given criteria $Q \subseteq P$. Let ϕ and ψ be two CTL_{-X} formulas with $scope(\psi) \subseteq P$.

$apn \models_{sf} \phi \Rightarrow apn' \models_{sf} \phi$.

$apn \models_{sf} \psi \Leftarrow apn' \models_{sf} \psi$.

Proof:

The theorem has been proved already see 4.3.1.

Lemma 4.3.3:
Let apn be a marked unfolded APN and Q be a slicing criteria such that $Q \subseteq P$. Let $apn = APNSlice(apn, Q)$ and $apn' = AbstractSlice(apn, Q)$. Let t be a neutral transition of the apn between p_1 and p_2. Let m and m' be two markings of apn and apn'. $p_1 \notin Q \wedge p_2 \notin Q \Rightarrow m'(p_1p_2) = m(p_1) + m(p_2)$, where $\{p_1, p_2\} \in P'$ and $m'(x) = m(x)$ for every $x \in P' \setminus \{p_1, p_2\}$.

Proof: Let t be a neutral transition. The markings of the places that are pre and post of a neutral transition are combined by abstract slicing algorithm (see Alg.11). By construction, it is guaranteed that the markings of a combined place in the abstract slice is equal to the sum of pre and post places of a neutral transition (in the APNsliced net) if pre or post places are not the criterion places. □

Theorem 4.3.3:
Let apn_0 be a marked unfolded APN and Q be a slicing criteria such that $Q \subseteq P$. Let $apn = APNSlice(apn, Q)$ and $apn' = AbstractSlice(apn, Q)$ be two sliced APNs. Let φ be a CTL formula.

$apn \models \varphi \Leftrightarrow apn' \models \varphi$.

Proof: We prove this theorem by contradiction. Let us assume to the contrary that $apn \models \varphi \Rightarrow apn' \not\models \varphi$. Intuitively, there exist a state (i.e., reachable markings) in the reachability graph that violates the property satisfaction. Let us assume that there exist such reachable marking m' in the abstract sliced APN that violates the property. There

are two possible cases to get such kind of markings. The first is to combine the places and the pre place of a neutral transition is the criterion place. The second is when the post place is the criterion place. Since, for both the cases, we can not combine the places if any of the pre or post places are in the criterion place by 4.3.3. Thus there does not exist any reachable state that violates the property in abstract sliced APN. So, we conclude that $apn \models \varphi \Rightarrow apn' \models \varphi$. Analogously, we can prove that $apn \models \varphi \Rightarrow apn' \models \phi$. □

4.3.6 Property Specific Slicing Algorithms

The idea of slicing to improve model checking is proved to be useful [CR94, LKCK00, Rak08, Rak11, Rak12, WCZX13, LOS+08, KR13]. The major challenge is to develop slicing algorithms that can further reduce the slice size. It is implicit that more the reduced slice size, the more reduction in state space. We propose to classify properties to develop more aggressive slicing algorithms. This classification is based on satisfaction of properties if they can already be determined or not by inspecting finite pre-fixes of traces of the transition system. We develop two different slicing algorithms based on our classification that are:

- **Safety Slicing**
- **Liveness Slicing**

4.3.7 Safety Slicing

Safety slicing has been defined as a static slicing algorithm. The idea of safety slicing was introduced by A.rakow in [Rak12] in the context of Petri nets (PNs). We adapt the idea and applied to the Algebraic Petri Nets (APNs) along with *neutral transitions*. The objective of safety slicing is to generate aggressive slicing algorithm (i.e., a slicing algorithm can generate smaller slice) by classification of properties. The safety slicing algorithm is only applicable to the safety properties and can not be applied to liveness properties. The reason why the slicing algorithm can produce smaller slice for safety properties is due to the fact that satisfiability of safety properties can already be determined inspecting finite prefixes of traces of the transition system.

4.3.8 The Slicing Algorithm: SafetySlicing

The safety 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 12: Safety slicing algorithm

SafetySlicing($\langle SPEC, P, T, F, asg, cond, \lambda, m_0\rangle, Q$){
$T' \leftarrow \{t \in T / \exists p \in Q \wedge t \in ({}^\bullet p \cup p^\bullet) \wedge \lambda(p,t) \neq \lambda(t,p)\}$;
$P' \leftarrow Q \cup \{{}^\bullet T'\}$;
$P_{done} \leftarrow \emptyset$;
while $((\exists p \in (P' \setminus P_{done}))$ **do**
  while $(\exists t \in ({}^\bullet p) \setminus T') \wedge \lambda(p,t) < \lambda(t,p))$ **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 Safety 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'. Only those transitions together with their incoming places are added that decrease the token count. Then the *neutral transitions* are identified and their *pre and post* places are merged to one place together with their markings.

Consider an example unfolded APN model shown in Fig.4.11**(A)** (Note: to present our idea about liveness slicing, we took an extremely simple version of unfolded APN model). Let us take an example safety property and apply *safety slicing* algorithm on the example unfolded APN model. Let us consider a property that place P_4 is never empty. Formally, we can write the property such as:

$\phi_3 = \mathbf{AG}(P_4 \neq \emptyset)$

Transitions T_1 is omitted because it does not decrease token counts on places P_1 and the it is not the criterion place as show in the Fig.4.12**(B)**. Places P_1 and P_2 are merged

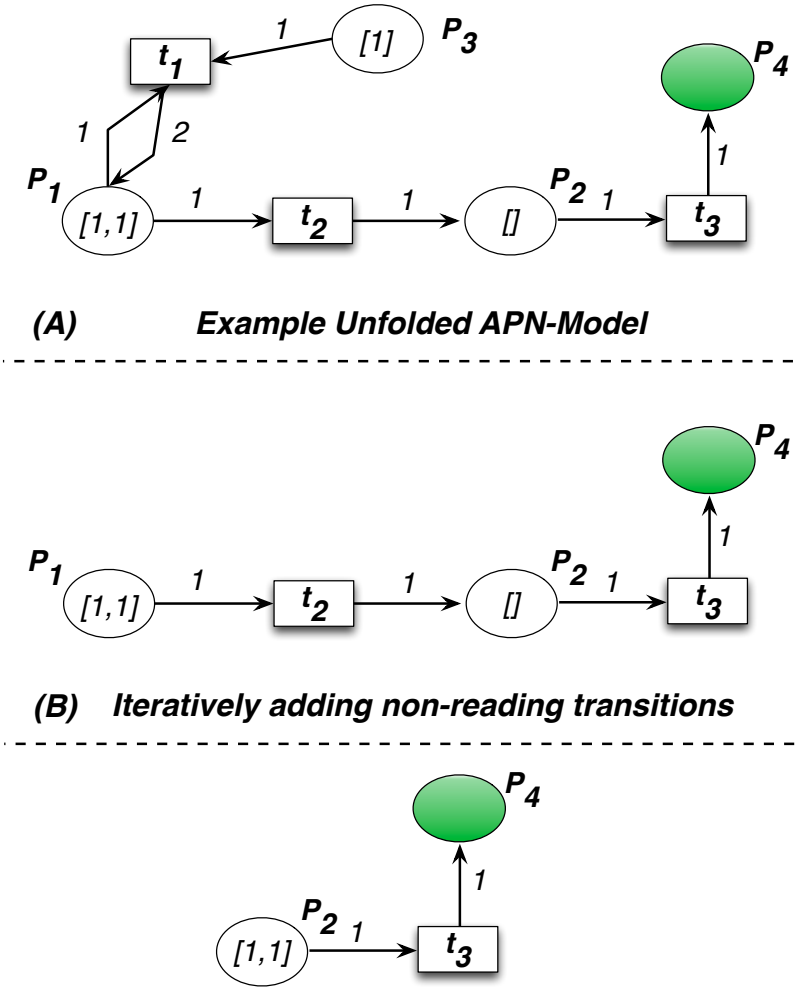


Figure 4.11: The sliced unfolded APNs (by applying *Safety slicing* algorithm)

together by identifying a neutral transition between them. The resultant sliced net as shown in the Fig.4.12**(C)** contains only one transition which is t_3. Let us now compare the number of states that are required to verify the given property ϕ_3 with and without safety slicing. The total number of states that will be generated are *9* without slicing whereas by applying slicing only *3* will be generated. We refer the interested reader to [Rak12] for the proof of preservation of safety properties by the *Safety slicing* algorithm.

4.3.9 Liveness Slicing

Liveness slicing has been designed as a static slicing algorithm. We notice that more aggressive slices can be generated if we design slicing algorithms for particular class of properties. In the previous section, we designed a slicing algorithm that is applicable to safety properties only. The idea of liveness slicing is similar to the safety slicing. The limitation of liveness slicing is that it preserves only particular liveness properties given by a formula $\exists\mathbf{F}(AP)$. The reason why this slicing algorithm generates smaller slice is that it does not include those transitions that decrease token count on places and these places are other than criterion places. Intuitively, we capture enough behaviors of the net that are required to verify the property.

4.3.10 The Slicing Algorithm: LivenessSlicing

The liveness 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 13: Liveness slicing algorithm

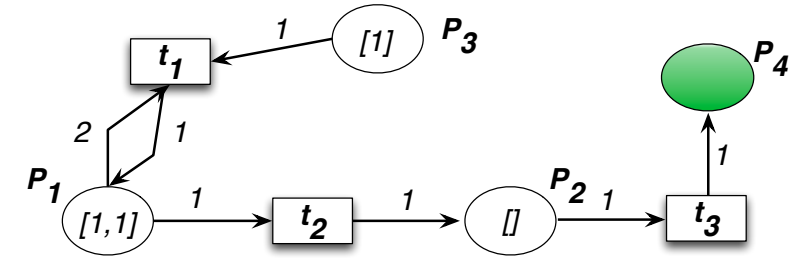
LivenessSlicing($\langle SPEC, P, T, F, asg, cond, \lambda, m_0\rangle, Q$){
$T' \leftarrow \{t \in T / \exists p \in Q \wedge t \in ({}^\bullet p \cup p^\bullet) \wedge \lambda(p,t) \neq \lambda(t,p)\}$;
$P' \leftarrow Q \cup \{{}^\bullet T'\}$;
$P_{done} \leftarrow \emptyset$;
while $((\exists p \in (P' \setminus P_{done}))$ **do**
  while $(\exists t \in ({}^\bullet p) \setminus T') \wedge \lambda(p,t) > \lambda(t,p))$ **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 Liveness 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'. Only those transitions together with their incoming places are added that increase the token count. Then the *neutral transitions* are identified and their *pre and post* places are merged to one place together with their markings.

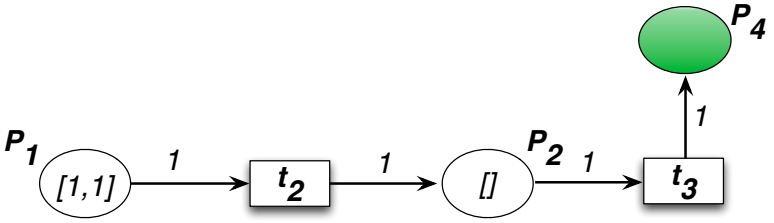
Consider an example unfolded APN model shown in Fig.4.12**(A)** (the APN model is quite similar to the one shown in Fig.4.11. The difference is that the transition t_1 produce less tokens than it consumes. Let us take an example liveness property followed by $\exists\mathbf{F}(AP)$ formula and apply liveness slicing algorithm on the example unfolded APN model. Let us consider a property that place eventually P_4 is not empty. Formally, we can write the property such as:

$\phi_4 = \exists\mathbf{F}(P_4 \neq \emptyset)$

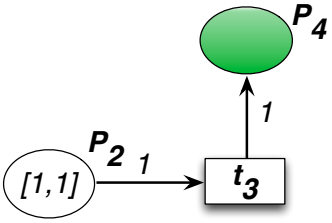
Transitions T_1 is omitted because it does not increase token counts on places P_1 and



(A) Example Unfolded APN-Model



(B) Iteratively adding non-reading transitions



(C) Identifying neutral transitions and merging places

Figure 4.12: The sliced unfolded APNs (by applying *liveness slicing*)

the it is not the criterion place as show in the Fig.4.12**(B)**. Places P_1 and P_2 are merged together by identifying a neutral transition between them. The resultant sliced net as shown in the Fig.4.12**(C)** contains only one transition which is t_3. Let us now compare the number of states that are required to verify the given property ϕ_3 with and without liveness slicing. The total number of states that will be generated are *9* without slicing whereas by applying slicing only *3* will be generated.

To show that the liveness slice preserves these particular properties (i.e., given by a particular formula $\exists\mathbf{F}(AP)$), it is sufficient to show that the sets of finite firing sequences of sliced and net and example net are equivalent. Intuitively, we can omit those transitions that do not decrease the token count of any place in the sliced net, so the token count on all places will be at least as low as it is on the example APN model's firing sequence.

4.4 Dynamic Slicing Algebraic Petri nets

Dynamic slicing refers to a particular slicing approach that utilizes initial markings. The objective of dynamic slicing is to generate a slice that can be used for debugging. Previously, we saw that static slicing is very useful in improving the state space for model checking. In dynamic slicing, we look for particular executions to see the token flows inside places. For example, a user can analyze a particular trace of marked Algebraic Petri net such that an erroneous state is reached. For dynamic slicing, we do not require an Algebraic Petri net to be unfolded.

4.4.1 The Slicing Algorithm: 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 information about the initial markings is utilized and it is directly applied to APNs instead of their

unfoldings.

Algorithm 14: 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
$T'' \leftarrow \{t \in T'/m_0[t\rangle\}$;
$P'' \leftarrow \{p \in P'/m_0(p) > 0\} \cup T''^\bullet$;
$T_{do} \leftarrow \{t \in T' \setminus T''/{}^\bullet t \subseteq P''\}$;
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 includes all the incoming transitions together with their input places until reaching a fixed 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. 4.13**(A)**, 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. 4.13**(B)**. The test input data can be generated for the sliced APN-model to observe which tokens are coming to the criterion place.

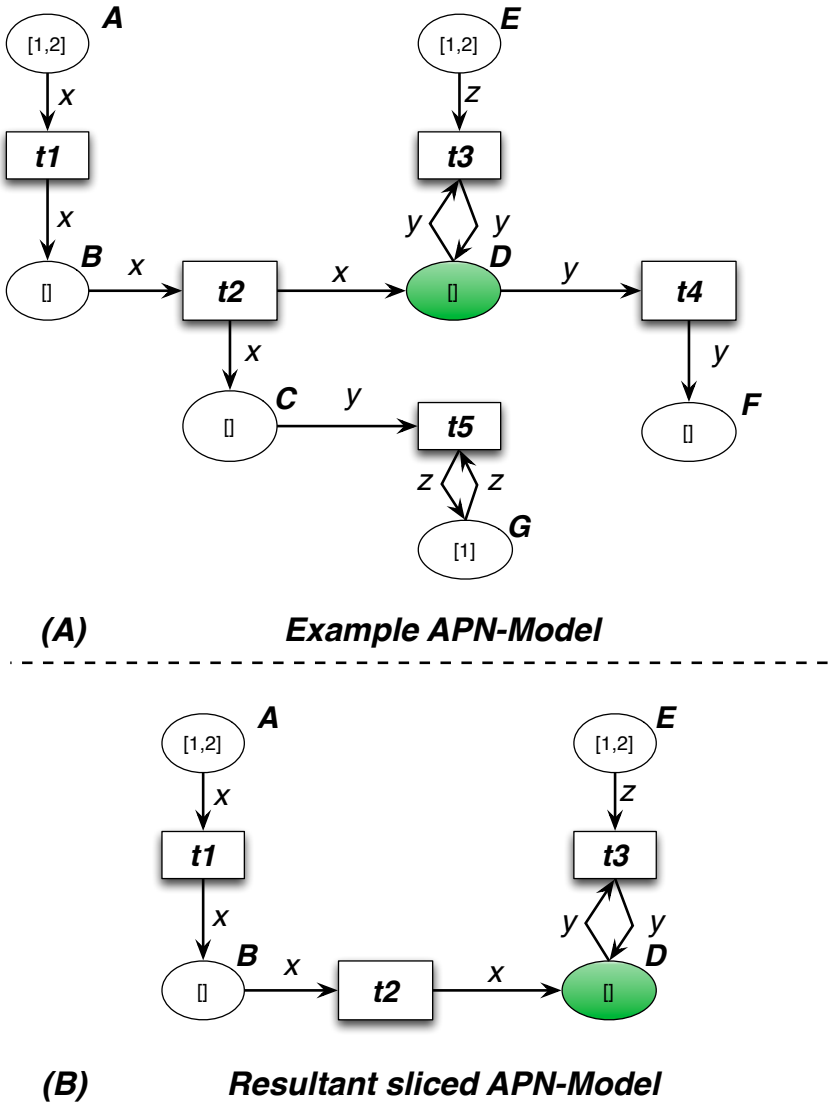


Figure 4.13: The sliced unfolded APNs (by applying *concerned slicing*)

4.4.2 Smart Slicing

Smart slicing is designed as a dynamic slicing algorithm. The objective is to bypass the state space generation or test input data generation. Consider for an example, if we are interested to know the values of tokens or number of tokens coming to a particular place that may erroneously produce tokens. We can determine the number of tokens and their values without generating state space by smart slicing algorithm.

4.4.3 The slicing Algorithm: Smart Slicing

The basic idea of smart slicing is to build a forward slice with respect to the criterion place. A forward slice can be computed by following Loren's slicing algorithm Alg.5. All the transitions together with their incoming places that are going to produce tokens to the criterion places are extracted. The benefit of forward slicing is that it does not include those transitions that can not be fired. The next step is to rename algebraic term with variables attached over the arcs of an APN-model. Starting from the root place, values are assigned to the variables from the initial markings of place. Then guard conditions, if there are any, are initiated. Values are passed to the next variable attached over the arc if the guard condition is true. By repeating this procedure for later places and transitions, we sum the markings of the criterion place.

Algorithm 15: Smart Slicing

Input: $(\langle S\,PEC, P, T, F, asg, cond, \lambda, m_0\rangle, Q)$.
Output: *Number of tokens and their values*.
1. *Compute forward slice for a given criterion place;*
2. *Rename terms with variables over the arcs that are going from a place to a transition (let us say* $x_0, x_1, ..., x_n$*) and from transition to a place (let us say y);*
3. *Select a root place from the set of places generated in step 1;*
4. *Repeat Until criterion place is reached;*
4.1. *Assign marking of root places to* $(x_0, x_1, ..., x_n)$*;*
4.2. *instantiate guard condition of transitions connected;*
4.3. *propagate the values of tokens from* $x_0, x_1, ..., x_n$ *to y;*
4.4. *sum the markings of selected place together with y;*
4.5. *Remove place from root places set;*
5. *Return number of tokens and their values in the criterion place*

Considering an example APN-model shown in Fig.4.14*(A)*, let us take an example place P_2 to know the number of tokens and their values coming to this place and apply our proposed smart slicing algorithm. First of all, a forward slice is generated by removing those transitions that can not be fired as shown in Fig.4.14*(B)*. The next step is to rename the algebraic term with variables over the arcs of transitions in a forward sliced net as shown in Fig.4.14*(C)*. In this case we rename the algebraic term x (i.e., over the arc from place P_1 to t_2) with x_0. And for the the algebraic term $x+1$ (i.e., over

the arc from transition t_2 to $P2$) we use y_0. Once the algebraic terms are renamed we can assign values to these variables from the initial markings which is $x_0 = 1$ as shown in Fig.4.14*(D)*. It is important to note that the choice of selecting values of tokens is non-deterministic. For an example in this particular case we can also assign $x_0 = 2$ instead of $x_0 = 1$. The next step is to instantiate guard conditions if any. In this case, we do not have any guard condition so it is set to true as shown in Fig.4.14*(E)*. Once the guard condition is true, an assigned value to the variable x_0 is passed to variable y_0 as shown in Fig.4.14*(F)*. A value to the variable y_0 is determined by computing its expression with the help of value coming from the variable x_0.

The next step is to assign the value of variable y_0 to the markings of a place attached to it. In this case, we add a token of value *2* to place P_2 as shown in Fig.4.14*(G)*. As a result we have successfully completed the first iteration and now for the other token value which is *2* the same procedure is followed as shown in the next step is to assign the value of variable y_0 to the markings of a place attached to it. In this case, we add a token of value *2* to place P_2 as shown in Fig.4.14*(H,I,J,K)*. Finally, we get the number of tokens and their values coming to the place P_2 without generating a state space. Obviously, smart slicing can refined further to investigate temporal properties as well.

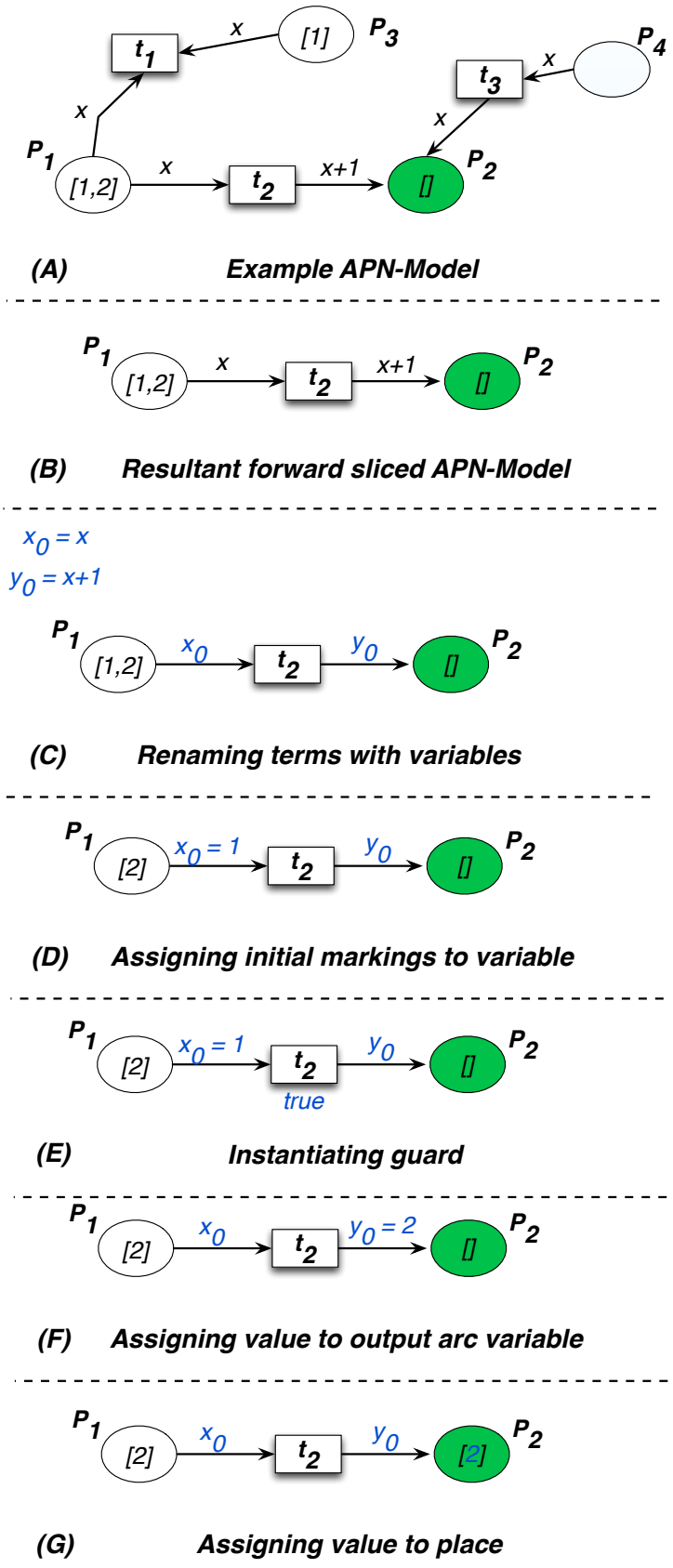


Figure 4.14: The resultant marking for token value 1 (by applying smart slicing)

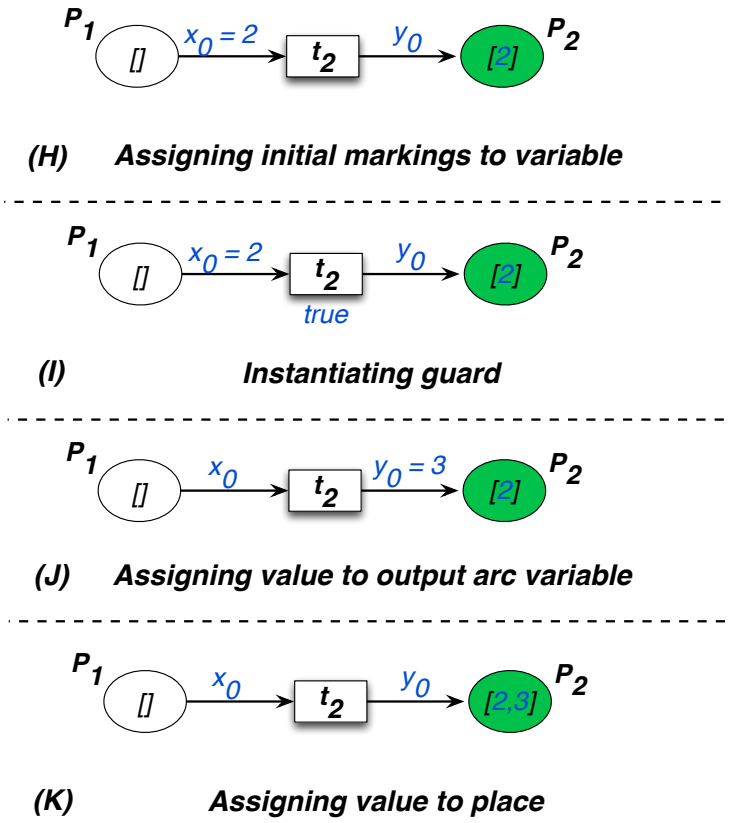


Figure 4.15: The resultant marking for token value 2 (by applying smart slicing)

4.5 Slicing Low-level Petri nets

We proposed several algorithms for slicing Algebraic Petri nets in this work. These algorithms can also be applied to low-level Petri nets with slight modification. The advantage of low-level Petri nets over high-level Petri nets is their simplicity and the availability of large number of tools to design and analyze them automatically. The slight modifications required to apply slicing algorithm designed for APNs refer to the syntactical changes in the algorithm but semantically there is no change. A λ function is used to compare arc inscriptions to identify reading or neutral transitions in Algebraic Petri nets. In low-level Petri nets, arc inscriptions are compared with a weight w function. The main change to apply slicing algorithms proposed for Algebraic Petri nets to low-level Petri nets is to replace λ function with w function. Obviously, the input function to generate slice will take a low-level Petri net instead of an APN. We argue that our proposed slicing algorithms give more refined results than the previous algorithms designed for low-level Petri nets. We select abstract slicing algorithm among other algorithms designed for Algebraic Petri nets and apply it to the low-level Petri nets. First of all, we need to redefine the notion of reading (resp.) non-reading transitions and neutral transitions in the context of low-level Petri nets. Fig 4.16, shows reading and neutral transitions for low-level Petri nets.

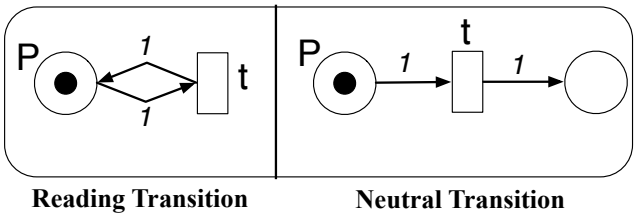


Figure 4.16: Reading and Neutral transitions in low-level Petri net

Reading (resp.) non-reading transitions are already defined for low-level Petri nets see definition 3.3.1. Let us define neutral transitions:

Definition 4.5.1: Neutral transitions of Petri nets
Let $t \in T$ be a transition in a PN. 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., $t \in T \wedge \exists p \exists p' / p \in {}^\bullet t \wedge p' \in t^\bullet \wedge |p^\bullet| = 1 \wedge |{}^\bullet t| = 1 \wedge |t^\bullet| = 1 \wedge w(t,p) = w(t,p')$.

4.5.1 The Slicing Algorithm: Abstract Slicing Algorithm for Low-level Petri nets

The abstract slicing algorithm starts with a Petri net model and a slicing criterion $Q \subseteq P$ containing crietrion place(s). We build a slice for an Petri net based on Q by

applying the following algorithm:

Algorithm 16: Abstract slicing algorithm for Low-level Petri nets

AbsSlicingPN($\langle P, T, f, w, m_0\rangle, Q$){
$T' \leftarrow \{t \in T / \exists p \in Q \wedge t \in ({}^\bullet p \cup p^\bullet) \wedge w(p,t) \neq w(t,p)\}$;
$P' \leftarrow Q \cup \{{}^\bullet T'\}$;
$P_{done} \leftarrow \emptyset$;
while $((\exists p \in (P' \setminus P_{done}))$ **do**
  while $(\exists t \in (({}^\bullet p \cup p^\bullet) \setminus T') \wedge w(p,t) \neq w(t,p))$ **do**
    $P' \leftarrow P' \cup \{{}^\bullet t\}$;
    $T' \leftarrow T' \cup \{t\}$;
  end
  $P_{done} \leftarrow P_{done} \cup \{p\}$;
end
while $(\exists t \exists p \exists p' / t \in T' \wedge p \in^\bullet t \wedge p' \in t^\bullet \wedge |{}^\bullet t| = 1 \wedge |t^\bullet| = 1 \wedge |p^\bullet| = 1$
$\wedge p \notin Q \wedge p' \notin Q \wedge w(p,t) = w(t,p'))$ **do**
  $m(p') \leftarrow m(p') \cup m(p)$;
  $w(t,p') \leftarrow w(t,p') \cup w(t,p)$;
  while $(\exists t' \in^\bullet t / t' \in T')$ **do**
    $w(p',t) \leftarrow w(p',t) \cup w(p,t')$;
    $T' \leftarrow T' \setminus \{t \in T' / t \in p^\bullet \wedge t \in^\bullet p'\}$;
    $P' \leftarrow P' \setminus \{p\}$;
  end
end
return $\langle P', T', f_{|P',T'}, w_{|P',T'}, m_{0_{|P'}}\rangle$;
}

In the Abstract slicing algorithm for low-level Petri nets, 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 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 an example Petri net model shown in fig. 4.17, 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_5 : *"The cardinality of tokens inside place P3 is always less than 5"*.

φ_6 : *"Eventually place P3 is not empty"*.

Formally, we can specify both properties using *CTL* as:

$\phi_5 = \mathbf{AG}(|m(P3)| < 5)$.

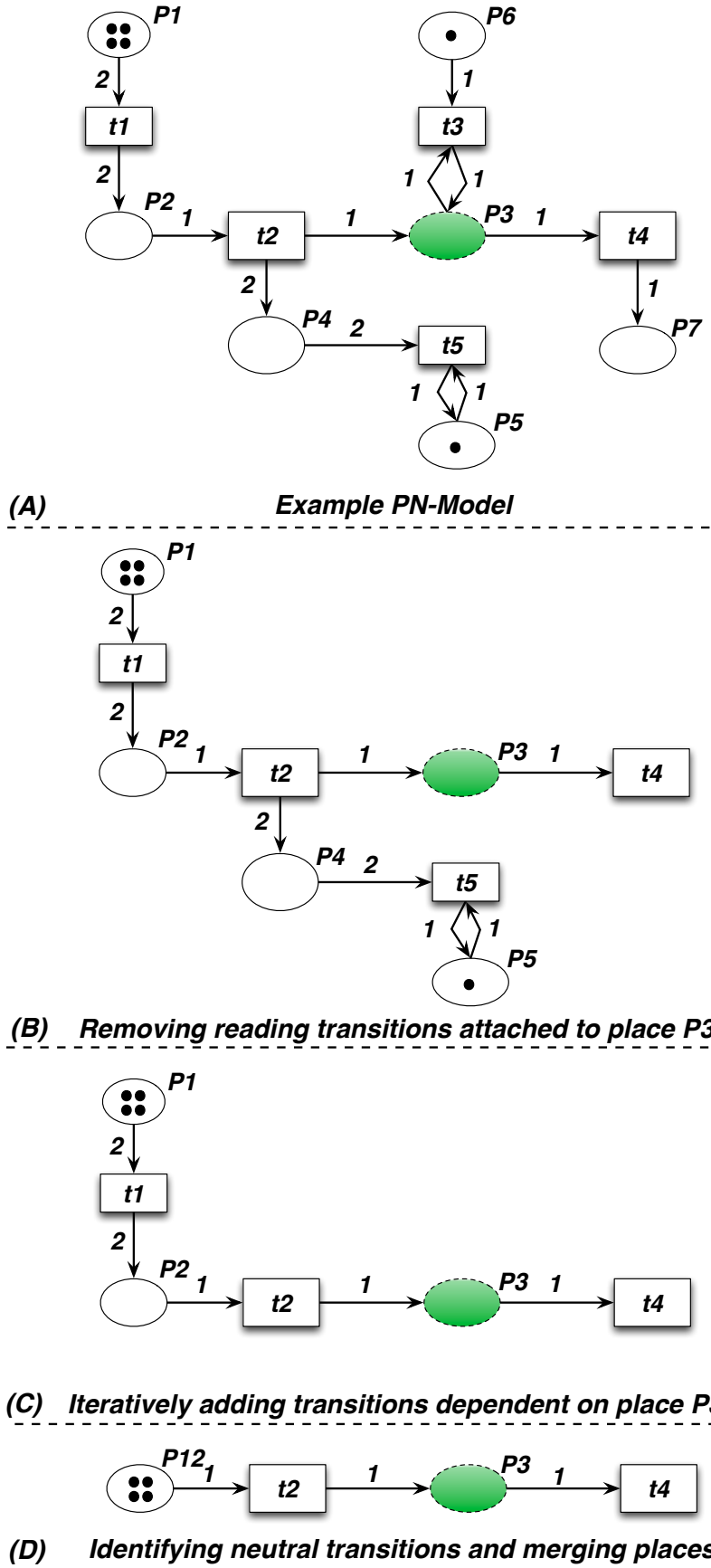


Figure 4.17: Petri net model and resultant sliced model after applying Abstract slicing algorithm

$\phi_6 = \mathbf{AF}(|m(P3)| = 1)$.

For both properties, the slicing criterion $Q = \{P3\}$, as *P3* is the only place concerned by the properties. The resultant sliced Petri net can be observed in fig.4.17***(D)***, which is smaller than the original Petri net.

Let us compare the number of states required to verify the given properties without slicing and after applying abstract slicing. The total number of states required without slicing is ***985***, whereas with the sliced model number of states is ***15***.

Chapter 5

Property Based Model checking of Structurally evolving Algebraic Petri nets

The true sign of intelligence is not knowledge but imagination.

— Albert Einstein

Iterative refinements and incremental developments is a commonly used technique for handling complex systems in hardware and software engineering. This involves creating a new specification or implementation by modifying an existing one [LB03, KR12]. In general, the modelers provide a first model that satisfies a set of initial requirements. Then the model can undergo several iterations or refinements until all the requirements are satisfied. In most cases, it is desirable for the developer to be able to assess the quality of model as it evolves.

The problem with the iterative and incremental development is that there is no guarantee that after each iteration or evolution of the model, it will still satisfy the previously satisfied properties.

Considering Algebraic Petri nets or Petri nets as a modeling formalism and model checking as a verification technique all the proofs are redone which is impractical. Most of the work regarding to improve the re-verification of evolving Petri nets is oriented towards the preservation of properties. Padberg and several other authors published extensively on the invariant preservation of APNs by building a full categorical framework for APNs (i.e. rule-based refinements [PGE98, LLV12, Er97]). Padberg considers the notion of a rule-based modification of Algebraic high level nets preserving the safety properties. The theory of a rule-based modification is an instance of the high-level replacement system. Rules describe which part of a net are to be deleted and which new parts are to be added. It preserves the safety properties by extending the rule-based modification of Algebraic Petri nets in contrast to transition preserving morphisms in [PGE98]. These morphisms are called the place preserving morphisms since they allow transferring specific temporal logic formulas expressing net properties from the source to the target net. Lucio et al presented a preliminary

study on the invariant preservation of behavioral models expressed in Algebraic Petri nets in the context of an iterative modeling process [LLV12]. They proposed to extend the property preserving morphisms in a way that it becomes possible to strengthen the guards without loosing previous behaviors.

In the previous chapter, we propose a slicing based solution to improve the model checking of Algebraic Petri nets or Petri nets. In this work, we propose a solution to improve the model checking of evolving system models expressed in APNs or PNs by re-using slicing techniques. Our proposal pursues three main goals. The first is to perform verification only on those parts that may affect the property a model is analyzed for. The second is to classify evolutions to identify which evolutions require re-verification. We argue that for a class of evolutions that require re-verification, instead of verifying the whole system only a part that is concerned by the property would be sufficient. The third goal is closely related to the previous proposals of property preserving evolutions. The theory of property preservation is based on the preservation of morphisms when evolving a system model. Our proposal improve the previous proposal by preserving morphisms restricted to the sliced part of a model.

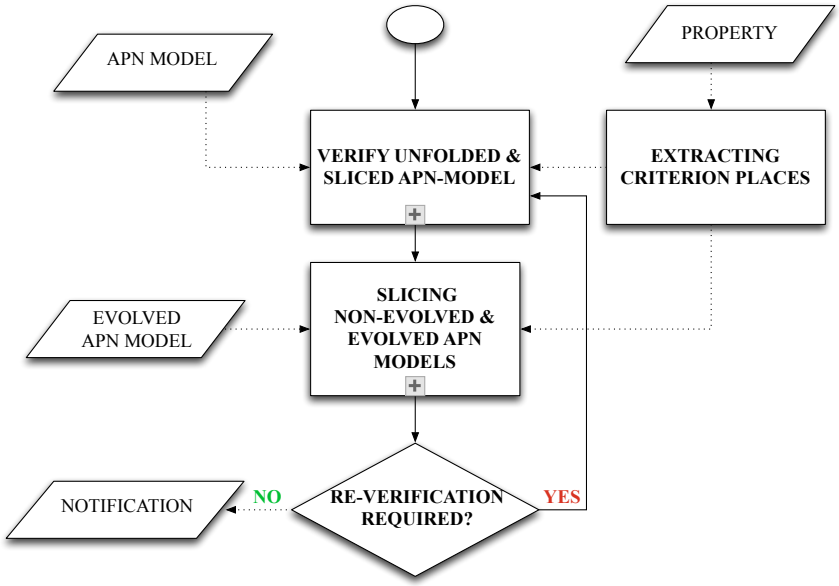


Figure 5.1: Process Flowchart of Property based model checking of Algebraic Petri nets

Fig.5.1, gives an overview using Process Flowchart of the proposed approach i.e., a slicing based verification of evolving Algebraic Petri nets. At first, verification is performed on the sliced APN model by taking properties into an account. Secondly, we build slices for evolved and non-evolved APN models. By comparing the resultant sliced models (i.e., APN and its evolved model), it is determined if an evolution has an impact on the property satisfaction and if it requires re-verification. In the worst case, if an evolution has an impact on the property satisfaction only the resultant sliced evolved Petri net model would be used for the verification. The process can be iterated

as per APN evolution.

5.1 Unfolding, Slicing Algebraic Petri nets

The first step in property based verification of structurally evolving Algebraic Petri nets is to verify the unfolded sliced APN model. As presented in the previous chapter, various slicing algorithms can be used to slice an APN model. In general, there are two types of slicing algorithms i.e., *static slicing* algorithms and dynamic slicing algorithms. The static slicing algorithms are designed to improve model checking whereas dynamic slicing algorithms are designed to improve testing. In the property based verification of structurally evolving Algebraic Petri nets, we consider only static slicing algorithms as we are interested to reason about the preservation of behavioral properties. The proposed static slicing algorithms are applied to an unfolded APN model. The unfolding helps to identify ground substitutions of algebraic terms used over the arcs of transitions in an APN model. This helps to generate a smaller slice as compared to applying slicing algorithm on APN model without unfolding.

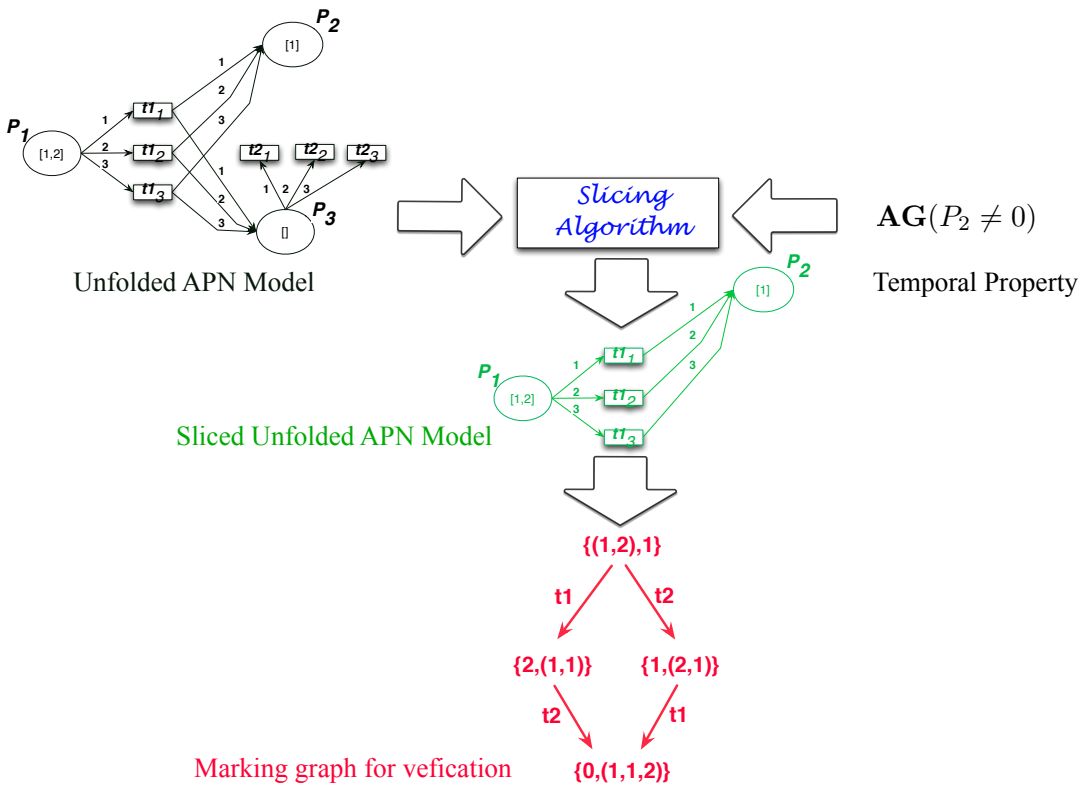


Figure 5.2: Overview of slicing Algebraic Petri nets

Fig.5.2 gives an overview of slicing an unfolded APN. First of all, a slicing algorithm takes APN model and temporal description of property to generate a slice. Then the

resultant sliced APN model is used to generate the marking graph (or state space) for the verification of property. For further details, we refer interested reader to the chapter 4.

5.2 Slicing evolved and non-evolved Algebraic Petri nets

The behavioral models of a system expressed in APNs are subject to evolve, where an initial version goes through a series of evolutions generally aimed at improving its capabilities. APN formalism consists of a control aspect that is handled by a Petri net and a data aspect that is handled by one or many AADT. In general, the evolutions of APNs can be divided into two parts,

- Evolutions of the structural aspect
- Evolutions of the data aspect

The evolutions that are taking place inside any of these aspects can disturb the property satisfaction. Remark that in this work, we made some assumptions that we allow only the structural evolutions while the data part and interesting properties are not subject to evolve. Informally, APNs evolutions of the control part can be changed such as: add/remove places, transitions, arcs, tokens and terms over the arcs. By notation, different APNs will be noted with superscripts such as $apn' = \langle SPEC, P', T', F', asg', cond', \lambda', m'_0 \rangle$.

Fig.5.3 ***(A)***, shows an example APN model, whereas Fig.5.3 ***(B)*** shows some possible structural evolutions that can happen to an APN model. As there is no guarantee that after every evolution of the APN model, it still satisfies the previously satisfied properties. A naive solution is to repeat model checking after every evolution, which is very expensive in terms of time and space.

To avoid repeated model checking, we propose a slicing based solution to reason about the previously satisfied properties. At first, by following our property based model checking proposal discussed in the previous chapter, we generate a partial state space to verify given properties. Consider an example unfolded APN model shown in Fig.5.4**(A)** (Note: we took an extremely simple version of unfolded APN model for the description of our proposed methodology). Let us take an example property, stating that the place P_4 is never empty. Formally, we can write the property such as:

$\phi_1 = \mathbf{AG}(P_4 \neq \emptyset)$

Afterwards, if there is an evolution of the APN model, we slice the evolved APN models with respect to the property by the same slicing algorithm (i.e. abstract slicing) used in the first step and compare both models for determining the satisfaction of property.

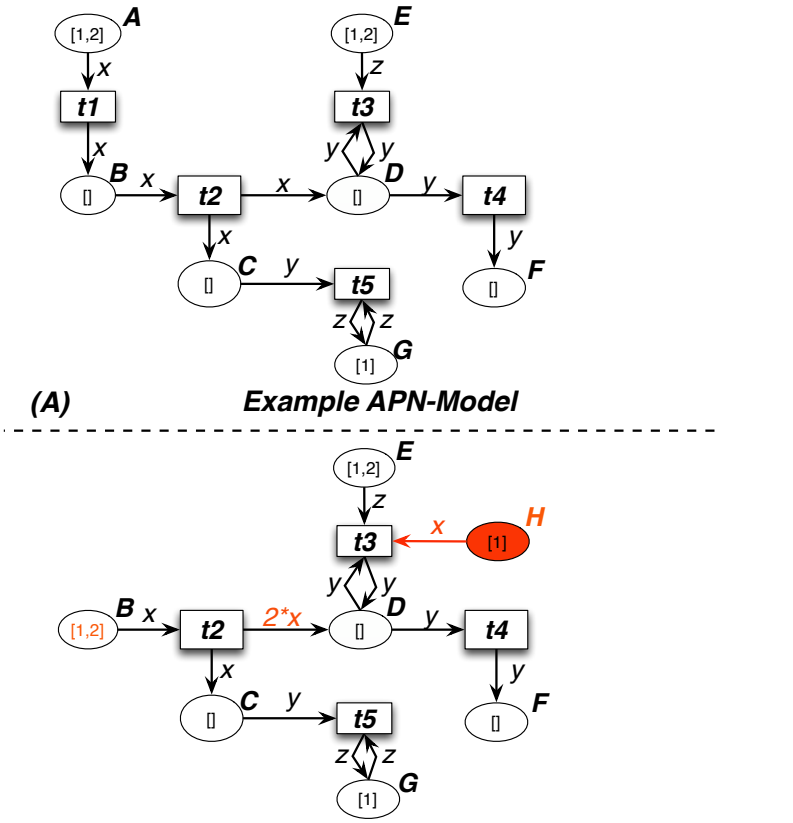


Figure 5.3: Structural evolutions to Algebraic Petri nets

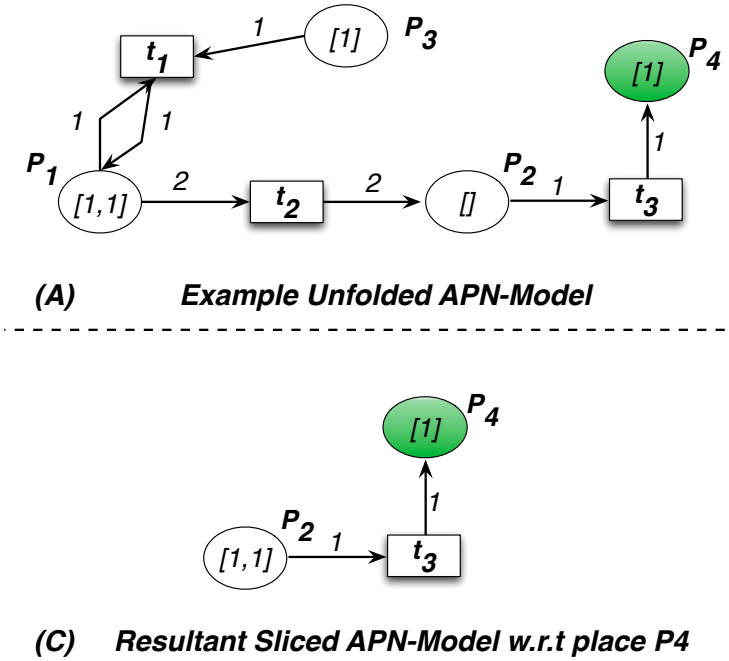


Figure 5.4: The resultant sliced APN-model after applying abstract slicing algorithm

5.2.1 Classification of Evolutions

As discussed above, we build slices for the evolved and non-evolved APN models with the help of *abstract slicing* algorithm. Once we built the slices, by comparing both APN models, we can divide the evolutions into two major classes as shown in the Fig.5.5. The evolutions that are taking place outside the slice and the evolutions that are taking place inside the slice. Further, we divide the evolutions that are taking place inside the slice into two classes i.e., the evolutions that disturb and those that do not disturb the previously satisfied property. We argue that the classification of evolutions helps to significantly reduce the re-verification cost and time.

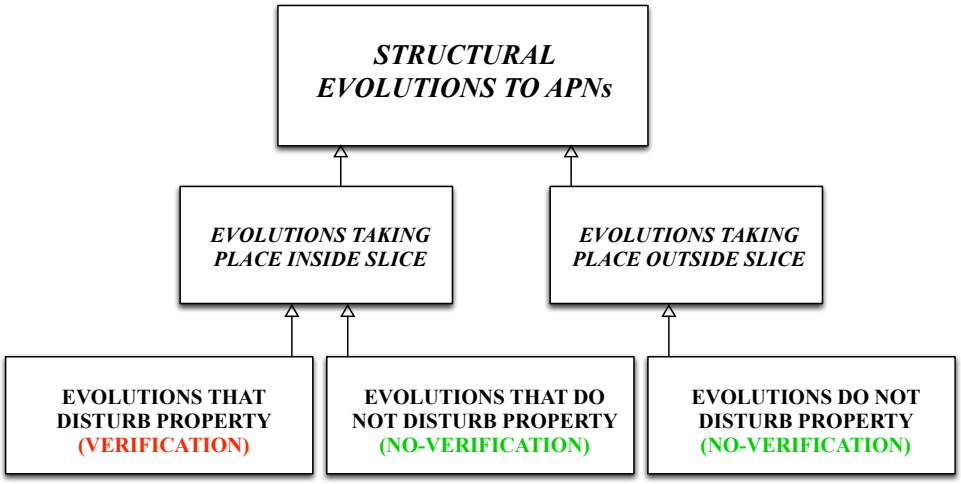


Figure 5.5: Classification of Structural Evolutions to Algebraic Petri nets

5.2.2 Evolutions taking place outside the Slice:

The aim of slicing is to syntactically reduce a model in such a way that at best the reduced model contains only those parts that may influence the property satisfaction. The rest of the model is discarded. Therefore, all the evolutions that are taking place outside the slice do not influence the property satisfaction. Consequently, model checking can be completely avoided for these evolutions. We formally specify how to avoid the verification if the evolutions are taking place outside the slice.

Theorem 5.2.1:
Let $apn_{sl} = \langle SPEC, P, T, f, asg, cond, \lambda, m_0 \rangle$ be a sliced unfolded APN model and $apn'_{sl} = \langle SPEC, P', T', f', asg', cond', \lambda', m'_0 \rangle$ be an evolved sliced unfolded APN model w.r.t the property ϕ. $apn_{sl} \models \phi \Leftrightarrow apn'_{sl} \models \phi$ if and only if

1) $P = P'$,

2) $T = T'$,

3) $f = f'$

4) $asg = asg'$

5) $cond = cond'$

6) $\lambda = \lambda'$

7) $m_0 = m'_0$

It is important to note that by assumption we consider structural evolutions, therefore algebraic specifications remain same for both nets. Informally, this theorem states that if an evolution takes place outside the slice then the evolved APN preserves the previously satisfied properties. It is important to note that we allow only structurally evolution to APN model and this is why algebraic specifications for both nets are same. Let us see the proof idea of this theorem.

Proof idea: According to the conditions imposed by theorem both the sliced net and evolved sliced net are same. Thus nothing to proof here.

To show, how practically we can identify such evolutions, let us recall the APN model and the example property given in the section 5.2. The example property is following $\phi_1 = \mathbf{AG}(P_4 \neq \emptyset)$. Fig.5.6 ***(B,C,D)*** shows some possible evolutions to the APN model that are taking place outside the slice. All the places, transitions and arcs that constitute a slice with respect to the property ϕ_1 are shown on the right side. In the first evolution example (i.e., shown in Fig.5.6 ***(B)***) a new place P_5 is added to the transition t_1. In the second evolution the place P_3 and transition t_1 is removed (i.e. shown in Fig.5.6 ***(C)***).

In the third evolution (i.e. shown in Fig.5.6 ***(C)***) tokens are added to the place P_3. For all such evolutions that are taking place outside the slice, we do not require re-verification because they do not disturb any behavior that may impact the satisfaction of the property.

5.2.3 Evolutions taking place inside the Slice:

For all the evolutions that are taking place inside the slice, we divide them into two classes

- Evolutions that require re-verification
- Evolutions that do not require re-verification

Identifying such class of evolutions is extremely hard due to non-determinism of the possible evolutions. Specifically, in APN small structural changes can impact the

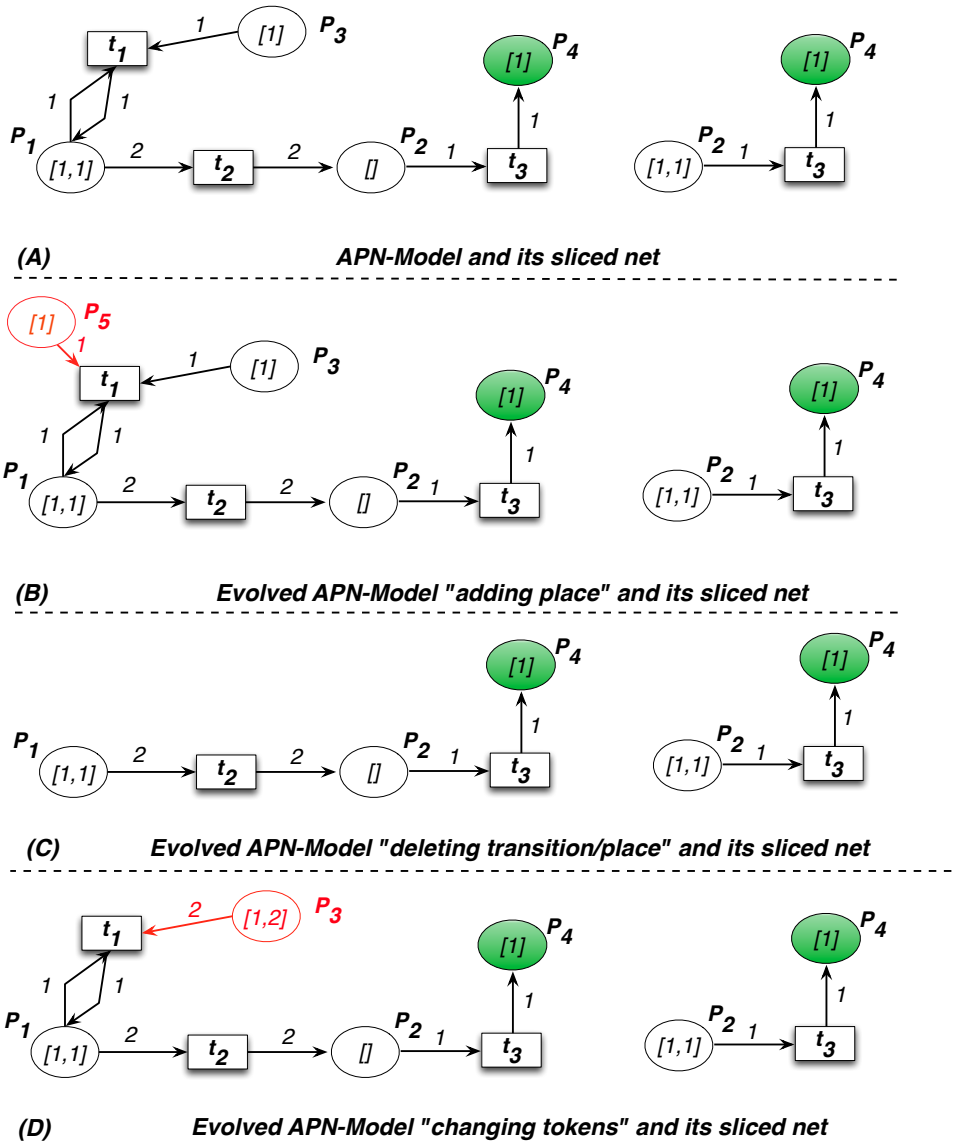


Figure 5.6: Structural evolutions to Algebraic Petri net model taking place outside the slice

behavior of the model. It is also hard to determine whether a property would be disturbed after the evolution or it is still satisfied by the model.

To identify evolutions that are taking place inside the slice and do not require verification, we propose to use the temporal specification of properties to reason about the satisfaction of properties with respect to specific evolutions. For example, for all the safety properties specified by the temporal formula $\mathbf{G}(\phi)$ *or* $\exists\mathbf{G}(\phi)$, if ϕ is an atomic formula, using the ordering operators $\leq$ *or* $<$ between the *places* and their *cardinality* or between tokens inside *places* and their *values*, then all the evolutions that decrease the tokens from places do not require verification because they do not impact the behavior required for the property satisfaction.

Theorem 5.2.2:
Let $apn_{sl} = \langle SPEC, P, T, f, asg, cond, \lambda, m_0\rangle$ be a sliced unfolded APN and $apn'_{sl} = \langle SPEC, P', T', f', asg', cond', \lambda', m'_0\rangle$ be an evolved sliced APN (in which tokens are decreased from the places) w.r.t the property ϕ. For all the safety properties specified by temporal formulas i.e., $\mathbf{G}(\phi)$ *or* $\exists\mathbf{G}(\phi)$, and ϕ a formula using $\leq$ *or* $<$ ordering operator between the places and their cardinality or tokens inside places and their values. $apn_{sl} \models \phi \Rightarrow apn'_{sl} \models \phi$ if and only if

1) $\forall p \in (P \cap P') \mid m_0(p) \geq m'_0(p)$,

2) $T = T'$,

3) $f = f'$

4) $asg = asg'$

5) $cond = cond'$

6) $\lambda = \lambda'$

7) $m_0 = m'_0$

Proof idea:

We can prove this theorem by contradiction. Let us assume that $apn_{sl} \models \phi$ and $apn'_{sl} \not\models \phi$. Let m *and* m' be two marking sequences of apn_{sl} and apn'_{sl} respectively. Since ϕ is a particular temporal formula, which is followed by an ordering operator $<$ *or* $\leq$. For the falsification of such formula in apn'_{sl}, there must exist a marking sequence such that $m' > m$, which is by construction (i.e., following conditions ***1,...,7*** of Theorem. 5.2.2) not allowed. Consequently, all the behaviors are preserved in the evolved apn'_{sl}. Hence $apn'_{sl} \models \phi$.

Let us recall the APN model given in the section 5.2. Let us take an example property $\phi_1 = \mathbf{AG}(|P_4| \leq 5)$ for which APN model is sliced and an example evolution to the APN model that is taking place inside the slice (shown in Fig.5.7 ***(B)***). All the places, transitions and arcs that constitute a slice with respect to the property ϕ_1 are shown on the right side. In the evolution example (i.e., shown in Fig.5.7 ***(B)***), tokens are

decreased from the place P_1.

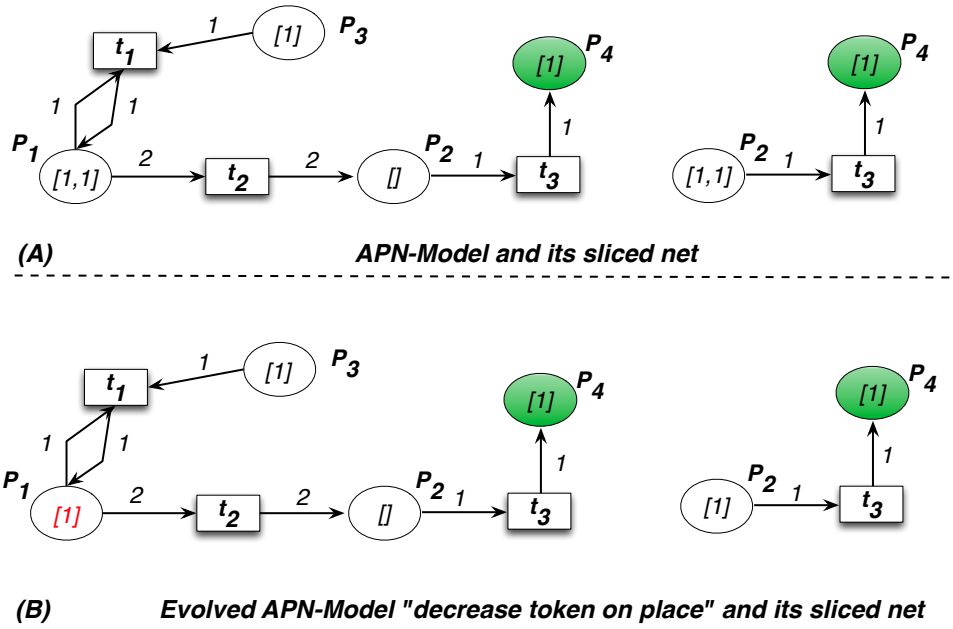


Figure 5.7: Structural evolutions to Algebraic Petri net model taking place outside the slice

For all such evolutions that are taking place inside the slice, we do not require re-verification because they do not disturb any behavior that may impact the satisfaction of the property.

For all the liveness properties specified by the temporal formula $\exists\mathbf{F}(\phi)$, and if ϕ a formula using the ordering operators ($\geq$ *or* $>$) the *places* and their *cardinality* or tokens inside *places* and their *values*, then for all the evolutions that increase the token count on the places, it is not required to re-verify them as they do not impact the behavior required for the property satisfaction.

Theorem 5.2.3:
Let $apn_{sl} = \langle SPEC, P, T, f, asg, cond, \lambda, m_0 \rangle$ be a sliced unfolded APN and $apn'_{sl} = \langle SPEC, P', T', f', asg', cond', \lambda', m'_0 \rangle$ be an evolved sliced apn' (in which tokens are increased from the places) w.r.t the property ϕ. For all the liveness properties specified by temporal formula $\exists\mathbf{F}(\phi)$, and ϕ is using the ordering operators $\geq$ *or* $>$ between the *places* and their *cardinality* or tokens inside *places* and their *values*. $apn_{sl} \models \phi \Rightarrow apn'_{sl} \models \phi$ if and only if

1) $\forall p \in (P \cap P') \mid m_0(p) \leq m'_0(p)$,

2) $T = T'$,

3) $f = f'$

4) $asg = asg'$

5) $cond = cond'$

6) $\lambda = \lambda'$

7) $m_0 = m'_0$

Proof idea: We can prove this theorem by contradiction. Let us assume that $apn_{sl} \models \phi$ and $apn'_{sl} \not\models \phi$. Let m *and* m' be two marking sequences of apn_{sl} and apn'_{sl} respectively. Since ϕ is a particular temporal formula, which is followed by an ordering operator $>$ *or* $\geq$. For the falsification of such formula in apn', there must exist a marking sequence such that $m' < m$, which is by construction (i.e., following conditions ***1,...,7*** of Theorem.5.2.3) not allowed. Consequently, all the behaviors are preserved in the evolved apn'_{sl}. Hence $apn'_{sl} \models \phi$.

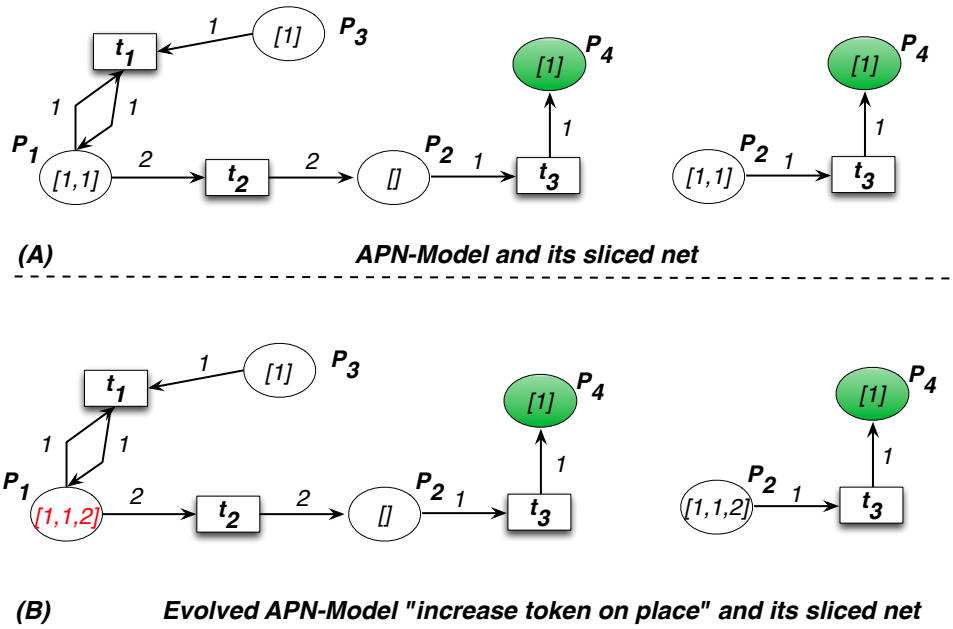


Figure 5.8: Structural evolutions to Algebraic Petri net model taking place outside the slice

Let us consider again the APN model given in the section 5.2, Let us take an example property $\exists\mathbf{F}(|P_4| > 3)$ for which the APN model is sliced. For some specific evolutions that are taking place inside the slice (in which tokens are increased in the places) , property remains true and we do not need re-verification (shown in Fig.5.8 ***(B)***).

We identified above that for several specific evolutions and properties, re-verification could be completely avoided, and for the rest of evolutions, we can perform verification only on the part that concerns the property. Even in the worst case, when evolutions are happening inside the slice, we can significantly improve the re-verification.

5.3 Property based verification of evolving low-level Petri nets

The proposed approach for Algebraic Petri nets (i.e., property based verification of structurally evolving Algebraic Petri nets) nets can be applied to low-level Petri nets directly. The main reason for the success of this low-level Petri net models is their simplicity in the description and a large number of tools exist to analyze them as compared to Algebraic Petri nets. The proposed approach becomes more simple to apply on low-level Petri nets as it does not require unfolding.

Fig.5.1, gives an overview using Process Flowchart of the proposed approach for a slicing based verification of evolving Petri nets. At first, verification is performed on the sliced Petri net model by taking a property into an account. Secondly, we build slices for evolved and non-evolved Petri nets models. By comparing the resultant sliced models (i.e., Petri net and its evolved model), it is determined if an evolution has an impact on the property satisfaction and if it requires re-verification. In the worst case, if an evolution has an impact on the property satisfaction only the resultant sliced evolved Petri net model would be used for the verification. The process can be iterated as per Petri net evolution.

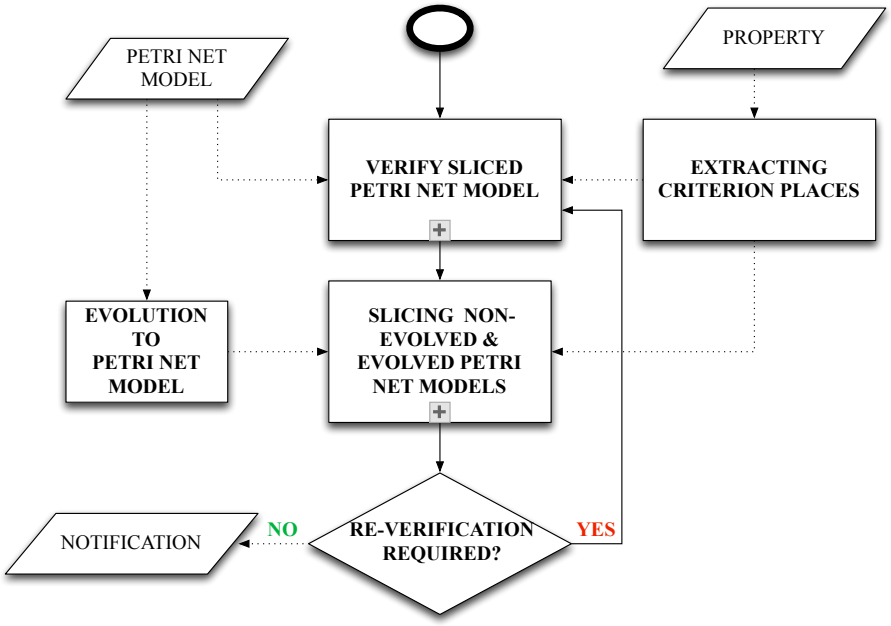


Figure 5.9: Process flowchart for verification of evolving Petri nets

Considering an example Petri net model shown in fig. 5.10, 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_3 : *"Eventually place P3 is not empty"*.

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

$\phi_3 = \mathbf{AF}(|m(P3)| = 1)$.

The slicing criterion $Q = \{P3\}$, since *P3* is the only place concerned by the properties. The resultant sliced Petri net can be observed in fig.5.10, which is smaller than the original Petri net.

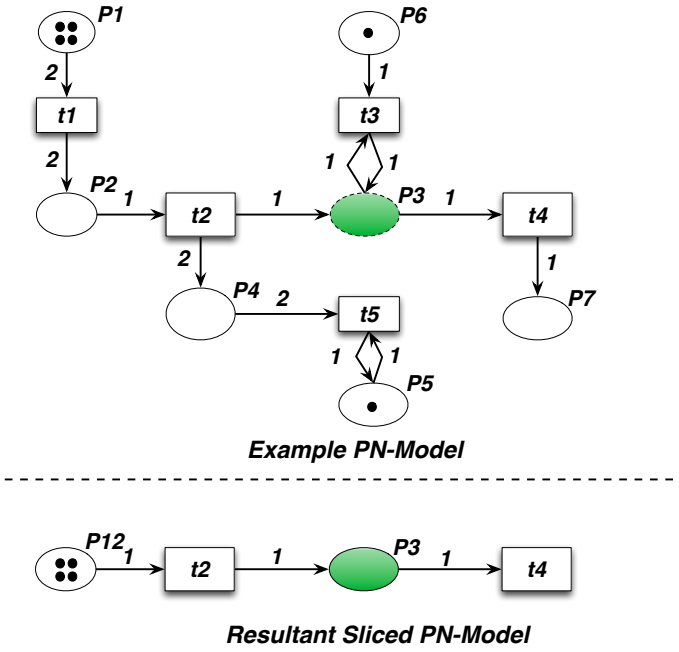


Figure 5.10: Example Petri net and its sliced net (by applying abstract slicing algorithm)

The second step is to allow different evolutions that can happen to the Petri net model shown in Fig.5.10 and build a slice for an evolved Petri net model. It is determined by comparing slices of evolved and the original Petri net model if re-verification is required. We present one example only i.e., when evolutions are taking place outside the slice.

Fig.5.11, shows some possible examples of the evolutions to Petri nets model that are taking place outside the slice. All the places, transitions and arcs that constitute a slice with respect to the property are shown with the blue doted lines (remark that we follow the same convention for all examples). In the example evolution, weight attached to the arc between transition *t2* and place *P4* is changed and shown with the red color. For all such kind of evolutions that are taking place outside the slice, we do not require verification because they do not disturb any behavior that may impact the satisfaction of the property.

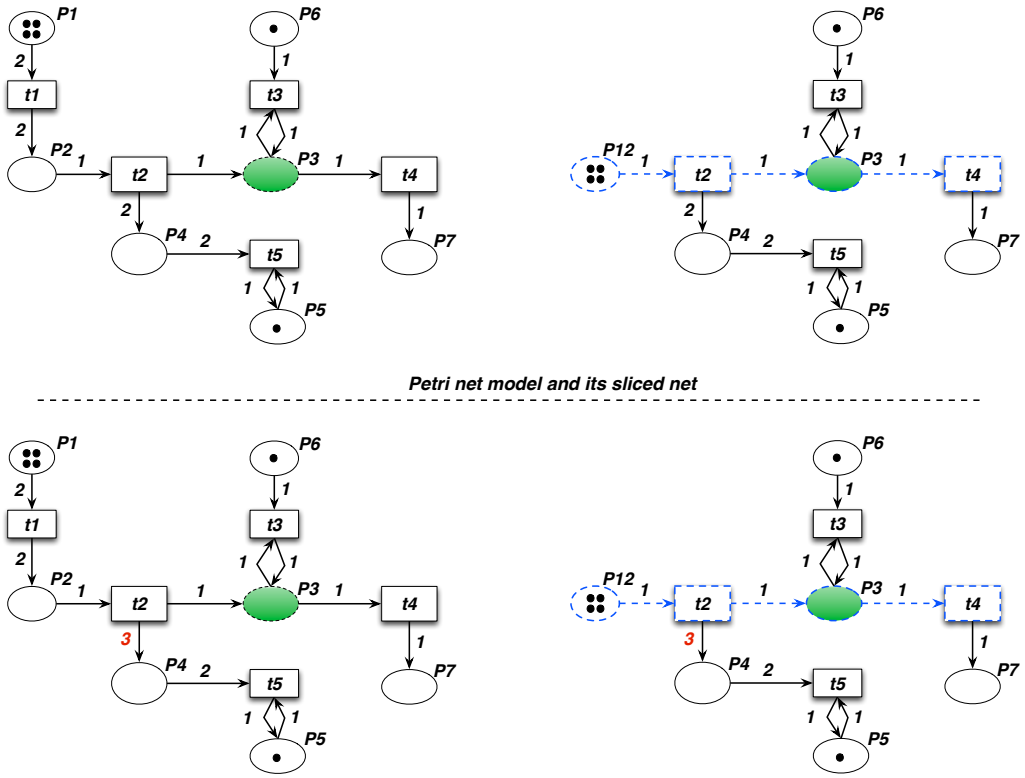


Figure 5.11: Evolutions to Petri net model taking place outside the slice

Chapter 6

Case Study & Evaluation

No amount of experimentation can ever prove me right; a single experiment can prove me wrong.

— Albert Einstein

Generally Crisis management System is the process by which an organization deals with a major event that threatens to harm the organization, its stakeholders, or the general public. Crisis management involves identifying, assessing, and handling the Crisis situation. The scope of this work is limited to one particular kind of Crisis management system, which is the Car Crash Crisis management system:

"A car accident or car crash is an incident in which an automobile collides with anything that causes damage to the automobile, including other automobiles, telephone poles, buildings or trees, or in which the driver loses control of the vehicle and damages it in some other way, such as driving into a ditch or rolling over. Sometimes a car accident may also refer to an automobile striking a human or animal" [KGM10].

In this work, we are using a particular kind of Car Crash management system. In order to to keep the case study manageable, we shall use a simplified model of Car Crash management system. We used textual use cases formalism for discovering and recording behavioral requirements.

6.0.1 Use Cases Car Crash Management System

In principle use-case scenario is a story about how someone or something external to the software (known as an actor) interacts with the system. The actors involved in our case study are:
Coordinator: A person in charge of recording the Crisis information.
System Administrator: An in charge person for managing Crisis.
SuperObserver: A skilled person dispatched to the crisis scene. In our case there are

two Superboservers which are fire fighter and lifter.

Use Case 1: Capture Crisis

Scope: Car Crash Crisis Management System Primary
Actor: Coordinator
Intention: The Coordinator intends to record a Crisis based on the information obtained from Capture data (Capture data can be a Fire on the Crisis location or Blockage of traffic).
Main Success Scenario:
Coordinator records Crisis and sends it to System.

1. Coordinator sends information to System as recorded.

Use case ends in success.

Use Case 2: Assign Mission

Scope: Car Crash Crisis Management System Primary
Actor: System Administrator
Intention: The System Administrator intends to assign a mission to Superobserver .
Main Success Scenario:
System Administrator assigns Superobserver to execute the mission.

1. System Administrator assigns a Crisis to Superobserver to execute Crisis mission.

Use case ends in success.

Use Case 3: Send Report

Scope: Car Crash Crisis Management System Primary
Actor: Superobserver
Intention: Send report to System after execution of the mission.

1. Superobserver sends report about executed Crisis mission.

Use case ends in success.

6.0.2 Formal Language Representation of Car Crash Management System

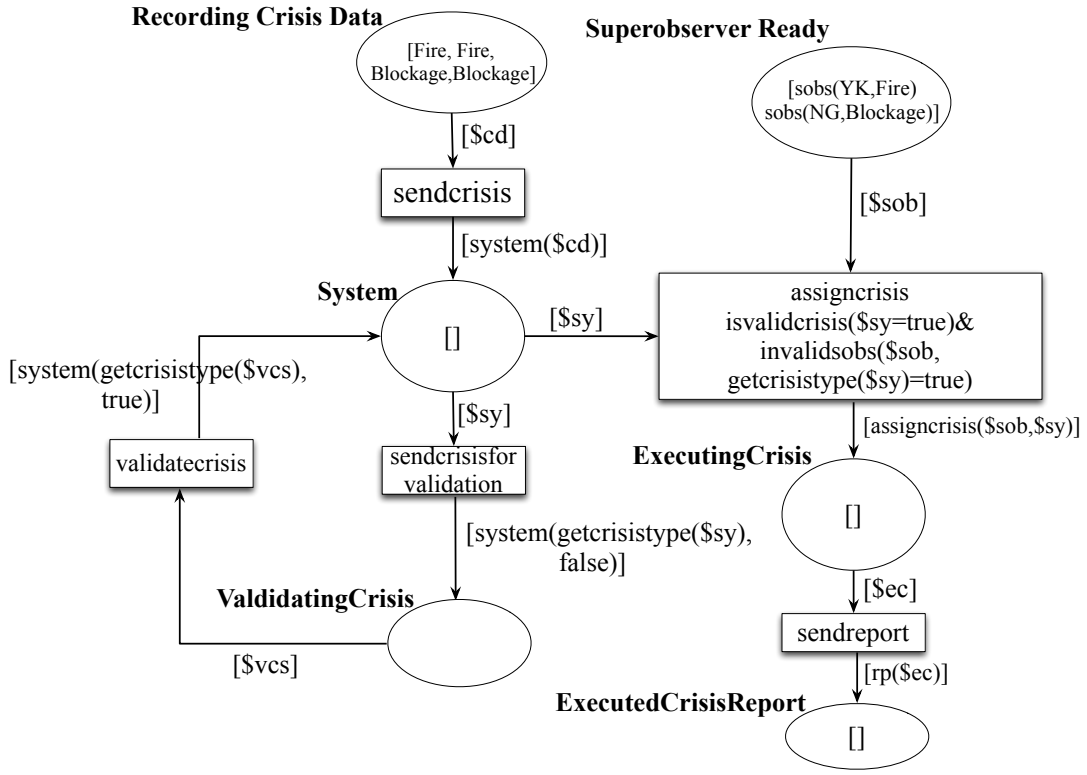


Figure 6.1: Car crash Algebraic Petri net model

The APN Model can be observed in Fig. 6.1, 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 recorded 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 `assigncrisis` has two guards, the first one is `isvalid(sy)=true` that enables to block invalid crisis reporting to be executed for the mission and the second one is `isvalid(sob,getcrisestype(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 [Kha12] for the algebraic specification of a car crash management system.

6.0.3 Interesting Properties

In formal verification, we verify that a system meets a desired property by checking that a mathematical model of the system meets a formal specification that describes the property. In general, there are two classes of properties i.e., safety properties and liveness properties. Intuitively, a property is a safety property if every violation occurs after a finite execution of the system. We can use this fact in order to base model checking of safety properties on a search for finite bad prefixes. Such a search can be performed using a simple forward or backward symbolic reachability check [KV01]. Informally, safety property of a system is a judgment of how likely it is that the system will cause harm to environment or people. Intuitively, liveness properties are considered as some thing good will eventually happen. A liveness property cannot be violated in a finite execution of a system because some thing good might occur at some time after execution ends [BBF+10].

An important safety threat, which we 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:

ϕ_1 : All the crises inside place `System` are validated eventually.

ϕ_2 : Place `Superobserver Ready` never contains more than two superobservers.

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.

$\phi_1 = \mathbf{AF}(\forall crisis \in System | isvalid(crisis) = true)$.

$\phi_2 = \mathbf{AG}(|SuperobserverReady| \leq 2)$.

In contrast to generate the full state space for the verification of both properties, we alleviate the state space by applying our proposed slicing algorithms. We have developed several slicing algorithms in this thesis which are *APNslicing, abstract slicing and liveness slicing*. Before applying slicing algorithm, we need to unfold Car Crash management system given in the Fig.6.1. The unfolded Car Crash APN model can be observed in Fig.6.2

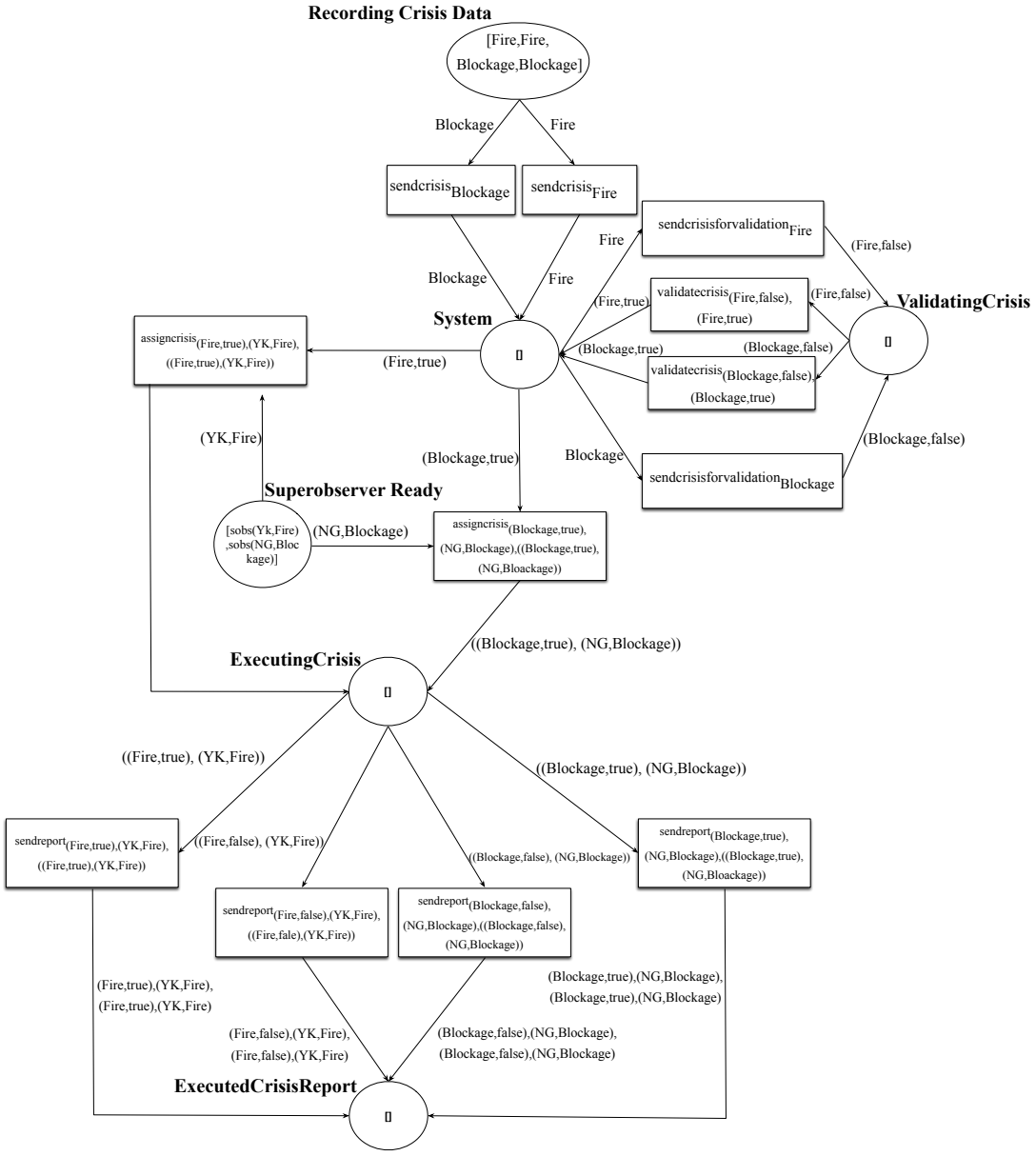


Figure 6.2: The unfolded Car crash Algebraic Petri net model

6.1 Applying Slicing Algorithms on Car Crash Management System

In general, slicing APN models is a pre-processing step towards model checking of properties. The sliced models are used to generate the state space to verify given properties instead of the original model. As discussed in in chapter 4, slicing algorithms are categorized in two major classes i.e., *static and dynamic* slicing algorithms. The objective of *static* slicing algorithms is to improve the model checking whereas *dynamic* slicing algorithms are used to improve the testing. In the chapter 4, we have developed several *static and dynamic* slicing algorithms. In this section, we shall select some of them to apply on the Car Crash Management system and compare the improvements in terms of model checking and testing. We shall also show the application of our proposed approach of property based model checking of structurally evolving APNs. The proposed approach is based on the slicing and by classifications of different structural evolutions and properties, it is determined whether re-verification is required or not (see details in chapter 5).

6.1.1 APNSlicing Algorithm on Car Crash Management System

The *APNSlicing* algorithm (given in the section 4.3.1) is an improved version of *Basic Slicing* algorithm (given in the section 3.2). In contrast to *Basic Slicing* algorithm, only *non-reading* transitions are included in the slice using *APNSlicing* algorithm. The *APNSlicing* algorithm takes an unfloded APN model and a set of criterion places. In the properties ϕ_1 and ϕ_2, the criterion places are `System and Superobserver Ready`. The *APNSlicing* 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 and ϕ_2. Respectively for ϕ_2, the algorithm starts from `Superobserver Ready` (as input criterion place) and builds the slice. The sliced unfolded car crash APN model is shown in the Fig. 6.3, for the properties ϕ_1 and ϕ_2. (Note: in this case study slices for both properties are same.) The places named `ExecutingCrisis` and `ExecutedCrisisReport` are removed together with a transition `sendreport`. The sliced Car Crash APN model can be used to generate the state space to verify the given properties.

Let us compare the number of states required to verify the given property ϕ_1 and ϕ_2 without slicing and after applying *APNSlicing* algorithm (given in Tab.6.6). The first column represents the Car Crash management system whereas in the second column properties are given. The total number of states that required to verify the given property is given in the third column. The fourth column represents the number of states on the sliced net after applying *APNSlicing* algorithm. The last column shows the reduction in percentage in comparison with the total number of states. (Note: we use the same conventions for the rest of slicing algorithms) We measure the effect of slicing in terms of savings of the reachable state space, since the size of the state space usually

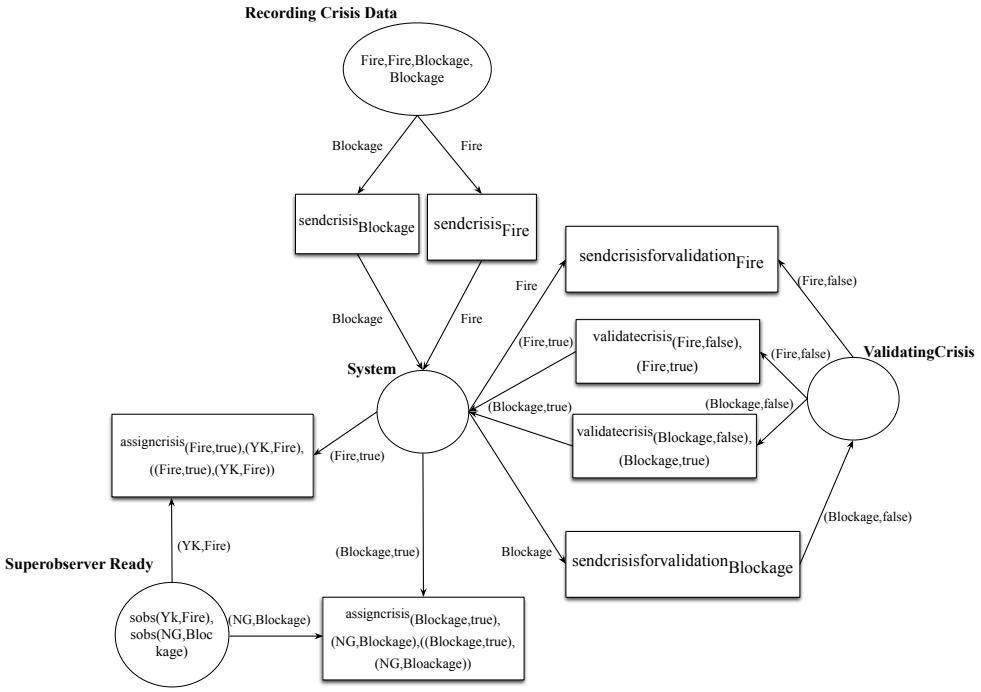


Figure 6.3: Sliced Car Crash APN model w.r.t ϕ_1 and ϕ_2 (by applying APNSlicing algorithm)

has a strong impact on time and space needed for model checking. The number of states required to verify ϕ_1 and ϕ_2 in a sliced net is very less in comparison with the actual number of states.

Table 6.1: Comparison of number of states with and without slicing

System	*Property*	*Tot.States*	*APNSlicing*	*Reduction*
Car Crash	ϕ_1	*324*	*196*	*39.51%*
Car Crash	ϕ_2	*324*	*196*	*39.51%*

6.1.2 Abstract Slicing Algorithm on Car Crash Management System

The *Abstract Slicing* algorithm (given in the section 4.3.3) is an improvement to the *APNSlicing* algorithm (given in the section 4.3.1). Along with including only *non-reading* transitions, *neutral* transitions are identified and their places are merged together. Similar to *APNSlicing* algorithm, 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 and ϕ_2. Respectively for ϕ_2, the algorithm starts from *Superobserver Ready (as input criterion place)* and builds the slice. The sliced unfolded car crash APN model is shown in the Fig. 6.4, for the properties ϕ_1 and ϕ_2. (Note: in this case study slices for both properties are the same.)

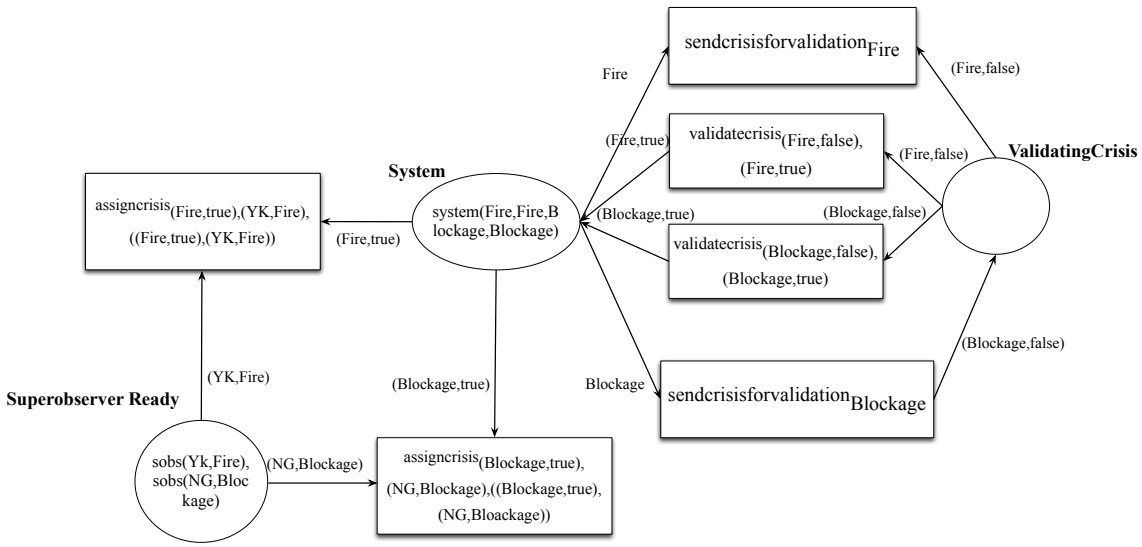


Figure 6.4: Sliced Car Crash APN model w.r.t ϕ_1 and ϕ_2 (by applying Abstract Slicing algorithm)

The place name `Recording Crisis Data` is merged with the place name `System` by removing `sendcrisis` transition (`sendcrisis` is identified as a *neutral transition*). The places named `ExecutingCrisis` and `ExecutedCrisisReport` are also removed along with a transition `sendreport`. The sliced Car Crash APN model can be used to generate the state space to verify the given properties.

Let us compare the number of states required to verify the given property ϕ_1 and ϕ_2 without slicing and after applying *Abstract Slicing* algorithm (given in Tab.6.7).

Table 6.2: Comparison of number of states with and without slicing

System	*Property*	*Tot.States*	*Abstract Slicing*	*Reduction*
Car Crash	ϕ_1	*324*	*144*	*55.56%*
Car Crash	ϕ_2	*324*	*144*	*55.56%*

Let us compare the number of states reduced by *APNSlicing* and *Abstract Slicing* by considering the same properties i.e., ϕ_1 and ϕ_2. In the table 6.8 shows the reduction column shows the improvement of *Abstract Slicing* as compared to *APNSlicing*.

Table 6.3: Comparison of number of states reduced by applying *APNSlicing* and *Abstract Slicing* algorithms

System	*Property*	*Tot.States*	*APN Slicing*	*Abstract Slicing*	*Reduction*
Car Crash	ϕ_1	*324*	*196*	*144*	*16.05%*
Car Crash	ϕ_2	*324*	*196*	*144*	*16.05%*

6.1.3 Concerned Slicing Algorithm on Car Crash Management System

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 produce 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 4.4.1). The sliced Car Crash APN model can be observed in the Fig 6.5. The places `Superobserver, ExecutingCrisis` and `ExecutedCrisisReport` and transitions `assigncrisis` and `sendreport` are removed as they are not producing any token to the criterion place.

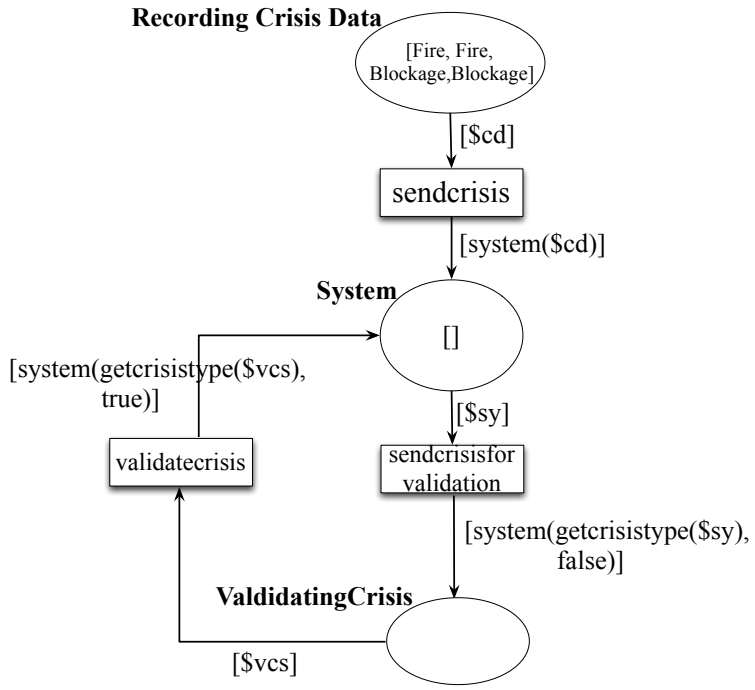


Figure 6.5: Sliced Car Crash APN model (by applying Concerned Slicing algorithm)

6.2 Structural Evolutions to Car Crash Management System

In general, verification is repeated after every evolution to determine whether a system satisfies previously satisfied properties or not. According to the proposed approach in chapter (), we can avoid repeated model checking by classification of evolutions and properties.

Let us take some evolution examples that can happen to Car Crash Management system and observe that whether the previously satisfied properties (i.e., ϕ_1 and ϕ_2) are still satisfied by the evolved model or not.

For example, Car Crash APN model shown in the Fig.6.1 evolves to restrict the type of reporting. A new guard for the transition `sendreport` is introduced to send reports of executed crises that are of type `Blockage`. The evolved Car Crash management system can be observed in Fig. 6.6.

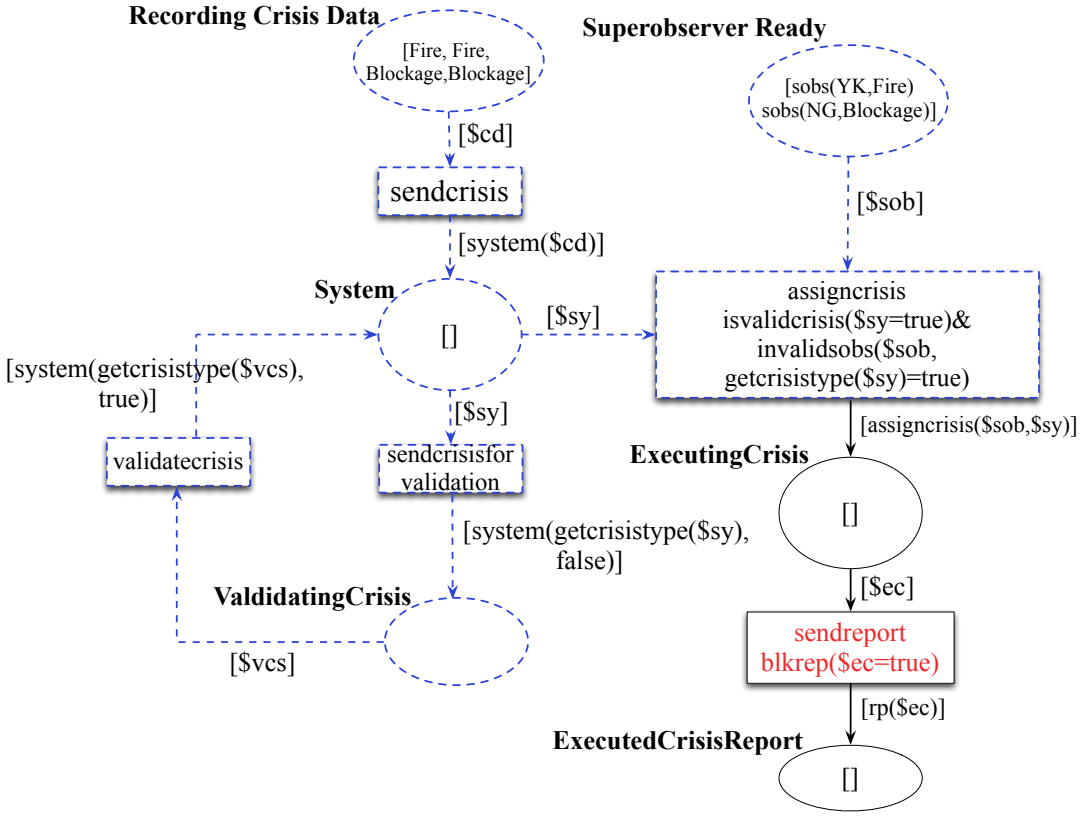


Figure 6.6: Evolved Car Crash (evolution taking place outside the slice)

All the places, transitions and arcs that constitute a slice with respect to properties ϕ_1 and ϕ_2 are shown with blue doted lines (Note: we use the same convention for the rest of evolved Car Crash APN model) examples . In the example evolution, a new guard is introduced which is shown by red color. According to the theorem proposed in the

section 5.2.2, for such evolutions that are taking place outside the slice, do not impact the satisfaction of previously satisfied properties. Consequently, model checking is completely avoided for such evolutions of Car Crash Management System and both properties ϕ_1 and ϕ_2 are preserved.

Let us take another example of evolution that is happening inside the slice of Car Crash Management system and observe the satisfaction of previous properties. The evolved Car Crash can be observed in Fig.6.7, in which tokens are removed from the place `Recording Crisis Data` and shown with the red color. According to theorem proposed in the section 5.2.3, for such particular evolutions that are taking place inside the slice and for particular properties such as ϕ_2, we do not require re-verification. For ϕ_1, we need to generate the state space of sliced Car Crash APN model.

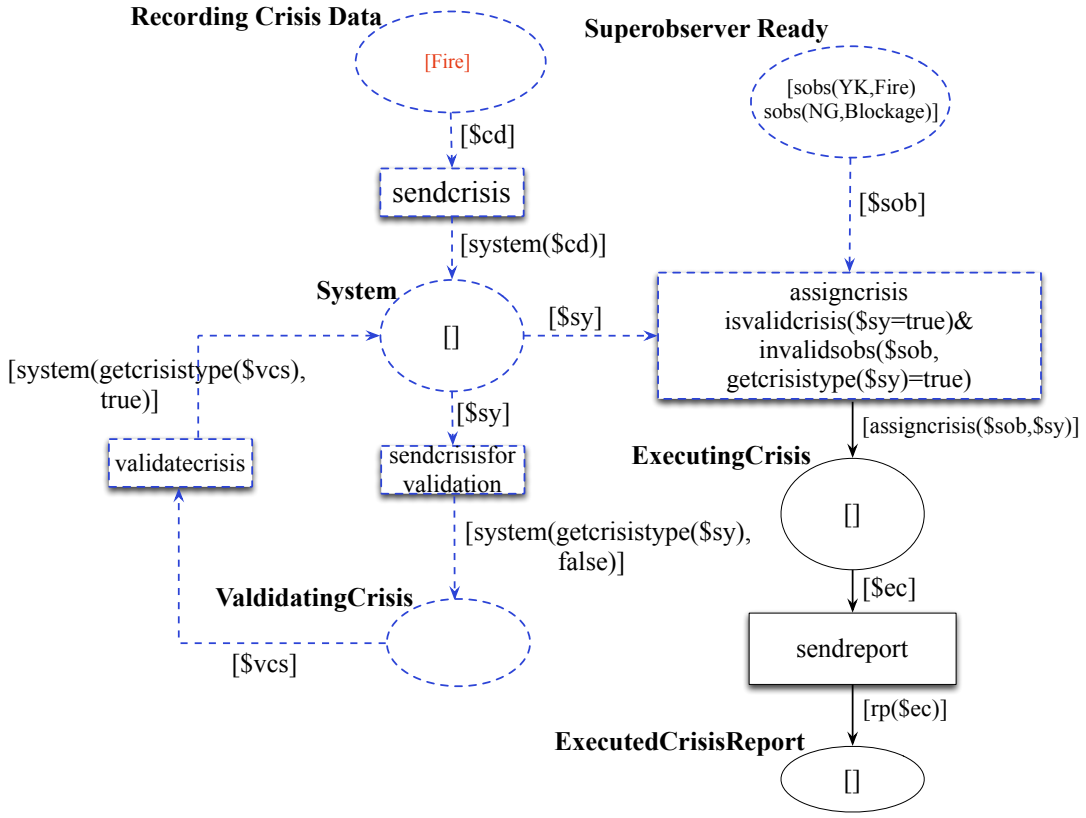


Figure 6.7: Evolved Car Crash (evolution taking place inside the slice)

Let us compare two examples of evolutions that are happening to Car Crash APN model with respect to the properties ϕ_1 and ϕ_2. For example, if we use *Abstract Slicing* algorithm, the table 6.9 shows the number of states reduced in the third column. The fourth column shows the type of evolution happening to Car Crash APN model.

Table 6.4: Comparison of evolutions and re-verification

Property	*Tot.States*	*Abstract Slicing*	*Evolution*	*Re-verification*
ϕ_1	*324*	*144*	*Evo1*	*YES*
ϕ_2	*324*	*144*	*Evo2*	*NO*

In the last column, based on the type of evolution and properties it is determined if re-verification is required or not.

6.3 Evaluation

In this section, we evaluate our proposed static slicing algorithms i.e. designed to improve the model checkings using existing benchmark case studies. The benchmark case studies are obtained from various sources such as PNs model checking context and previous research articles (which are used to evaluate slicing algorithms) []. We collected different examples to cover a wide range of systems with different characteristics.

Table 6.6 shows the case studies together with the size of APN model with respect to the number of places, transitions and arcs (i.e., shown in second, third and fourth columns). The last column indicates if either the net type is strongly connected or weakly connected. The net size of bench mark case studies when represented in APNs is very small and easy to understand as compared to PNs.

In general, every slicing algorithm takes criterion places and APN/PN model as an input. The criterion places are extracted from the temporal description of properties. One difficulty in evaluating slicing is to determine the interesting properties about the model. In principle, properties could involve any places in the model. To overcome this difficulty, we perform evaluation in two ways i.e.,

- **Generate slices for every place**
- **Generate slices for practically relevant properties**

Generate slices for every place: The first way to avoid the difficulty of determining the relevant properties is to generate slices for every place. Perhaps, it is not necessary that every place corresponds to a meaningful property. We follow [Rak11] to assume that they are not interesting properties to initially marked places and abandon them.

Generate slices for practically relevant properties: The second way is to select practically relevant properties and to show that the state space could be reduced for them by slicing. We took some specific examples of temporal properties from the APN models given in the Tab.6.6.

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. Instead of presenting case studies for which our proposed slicing algorithms work best, it is also interesting to see case studies average or worst results. We now present a comparative evaluation on the benchmark case studies.

A lot of model checkers has been designed to analyze Petri net models [BHMR10, Mä02, JKW07, BJS09, BV06]. A complete list of model checkers designed in the context of all classes of Petri nets can found in [Pet]. The common objective of every model checker is to explore all the possible states to check property satisfaction. It is important to note that the existing model checkers are domain specific and are not

general enough to consider all the proposed classes of Petri nets. Among all the existing model checkers for Petri nets, majority of them consider low-level Petri nets. The process of model checking Petri nets consists of three steps as show in Fig.6.8. The first step is to design a Petri net model by a model checking tool (i.e., for which class of Petri nets the model checker is designed). The second is to specify desired properties in some logical formalism such as temporal logic. In general, a model checker provides graphical editor to draw Petri nets models and a property language.

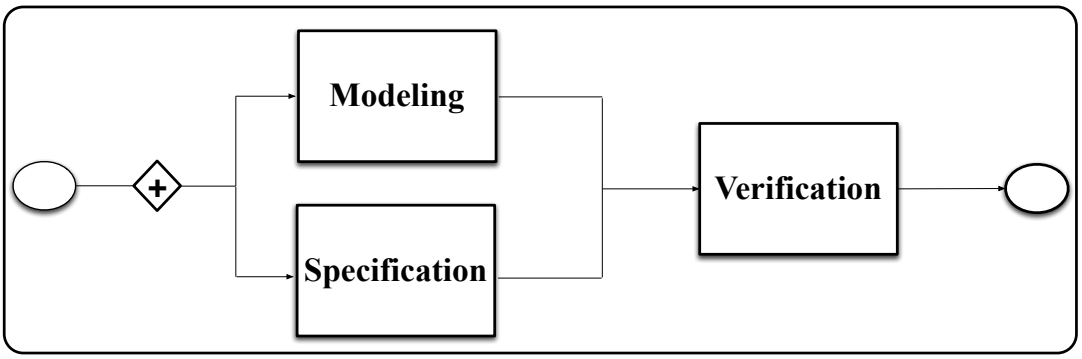


Figure 6.8: Model checking process

Different model checkers can vary with respect to their modeling domain and the properties they can verify. For an example *AlPiNA* model checker takes only APN models as an input [BHMR10]. Specifications in *AlPiNA* are composed of two parts: an algebraic specification, which is a set of abstract definitions of sorts and associated operations. The second is a Petri net, which is represented graphically. *AlPiNA* is able to decide on the satisfaction of invariant properties on those nets. The invariants are expressed as conditions on the tokens contained by places in the net at any state of the net semantics. Invariants are built using first order logic, the operations defined in the algebraic specifications and additional functions and predicates on the number of tokens contained by places.

Another known model checker in the family of Petri nets is CPN Tools (i.e., a tool for editing, simulating, and analyzing Colored Petri nets)[JKW07]. The tool was developed by the CPN Group at Aarhus University from 2000 to 2010. The tool also supports the basic Petri nets plus timed Petri nets. It has a simulator and a state space analysis tool is included. It has a feature to perform incremental syntax checking and code generation, which take place while a net is being constructed. Table 6.5, enlists some of the existing model checkers for different classes of Petri nets. In the first column, the name of model checker is given in abbreviation whereas the second column states the class of Petri nets for which it works and finally class of properties are given that can be verified by using temporal formulas or propositional logic formulas.

In general, the slicing process proceeds in three steps as shown in Fig.6.9. First of all an APN/PN model is drawn along with temporal formulas as properties. From the temporal formulas criterion places are extracted. Finally, a slicing algorithm takes criterion places APN/PN model to generate a sliced model. We use our implemented

Table 6.5: Different Model Checker for Petri nets Classes

Model checker	*Class of PNs*	*Properties*
AlPiNA	*APN*	*invariants*
CPN Tools	*CPN/PN*	*boundedness & liveness*
MARIA	*HLPN/PN*	*safety & liveness*
TINA	*TPN*	*safety & liveness*
TAPAAL	*TPN*	*safety & liveness*

tool SLAP$_n$ to draw the APN/PN model and slice it (see details in chapter 7). In the tool a user friendly environment is provided to draw an APN or PN model and different slicing algorithms are implemented. Secondly, after having a sliced APN model, a model checker is used to generate the state space. Another way to perform slicing is to use any existing editor to draw APN or PN model and then apply slicing algorithms on the generated net. Once a sliced net is obtained, a model checker can be used to generate the state space.

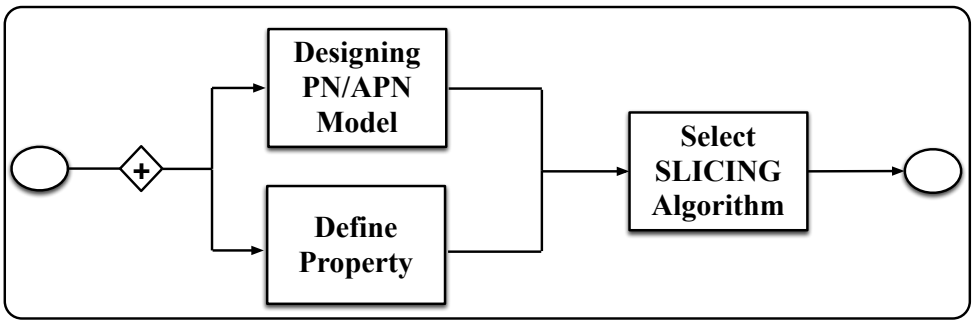


Figure 6.9: SLAPn Overview

Table 6.6: Benchmark Case Studies APN models

System	*No.Places*	*No.Transitions*	*No.Arcs*	*Init.Marking*	*Net type*
Complaint Handling	*10*	*9*	*20*	$RecComp = \{1, 2, 3\}$	*Weak.Connect*
House Construction	*26*	*18*	*51*	$p_1 = \{1, 2, 3\}$	*Weak.Connect*
Divide & Conquer	*23*	*19*	*47*	$p_1, p_5, p_{12}, p_{19} = \{1, 1, 1\}$	*Weak.Connect*
Flexible Manufacturing System	*22*	*20*	*50*	$p_1, p_8, p_{12}, p_{20} = \{1, 1, 1, 1, 1\}$, $p_{11} = \{1\}, p_5 = \{1, 1, 1\}, p_{18} = \{1, 1\}$	*Weak.Connect*
Beverage Vending & Machine	*5*	*3*	*7*	$idle = \{1, 2, 3\}, drinks = \{1, 2, 3\}$	*Weak.Connect*
Insurance Claim	*16*	*19*	*39*	$p_1 = \{1, 2\}$	*Weak.Connect*
A Customer support Production system	*17*	*11*	*33*	$p_1 = \{1, 1\}, p_2 = \{1, 1\}, p_4 = \{1\}$	*Weak.Connect*
Car Crash Management system	*6*	*5*	*11*	$RecCrisData, SuperObsReady = \{1, 2\}$	*Weak.Connect*
Electronic Trade System	*20*	*17*	*42*	$p_1 = \{1, 2\}, p_7 = \{1\}$	*Weak.Connect*
Daily Routine of 2 Employees & Boss	*10*	*11*	*34*	$A_1 = \{1, 2\}, B_1 = \{1\}$	*Str.Connect*
Simple Protocol	*7*	*5*	*12*	$PacketToSend = \{1, 2\}, NS = \{1\}$	*Str.Connect*
Producer Consumer	*5*	*4*	*12*	$prod = \{1, 2\}, cons = \{1\}$	*Str.Connect*
Kanban	*16*	*16*	*40*	$p_8, p_{26}, p_9, p_{14} = \{1, 1\}$	*Str.Connect*
Dining Philosophers	*6*	*4*	*13*	$think = \{1, 2\}$	*Str.Connect*

6.3.1 Applying slicing algorithm to generate slices for every place

In this section, we generate the slices for every place in the benchmark case studies APN models and present the impact of slicing algorithm in terms of state space. We applied only two slicing algorithms i.e., *APNSlicing* and *AbstractSlicing*. The reason to select these two algorithms is that they preserve more general class of properties as compared to others. In the next section, other algorithms are also used to generate slices for practically relevant properties.

For a fair comparison, we divide the results of slicing algorithm into three cases

- **Best case:** it refers to the best possible reduction achieved among all the possible properties.
- **Average case:** it refers to the average reduction achieved for all the possible properties.
- **Worst case:** it refers to the situation when there is no reduction.

First of all, we applied *APNSlicing* algorithm on every place of bench mark case studies APN models. Let us study the results summarized in the Table.6.7, the first column shows different APN models under observation. Based on the initial markings, the total number of states is shown in the second column. Best reduction and average reduction (shown in the third and fourth column) refers to the biggest and an average achievable reduction in the state space among all possible properties. The fifth column reports how many places (related to the properties) in the model lead to no reduction using our slicing method. Finally, the structure of the APN models under observation is given. Results clearly indicate the significant improvement in reducing the state space by slicing algorithms, and the proposed *APNSlicing* algorithm can alleviate the state space even for some strongly connected nets.

Secondly, we applied *Abstract Slicing* algorithm on every place of bench mark case studies APN models as shown in the Table.6.8.

Let us compare the results of both algorithms (i.e., *APNSlicing* and *Abstract Slicing* algorithms) with each other to see their performance in terms of state space reduction with different case studies. Let us study the graph shown in the Fig.6.10, along the *y-axis* of the graph bench mark case studies are listed whereas along the *x-axis* number in percentage are given to compare the reduction impact. We took two parameters from the previous Tables.6.7, 6.8 i.e., average case reduction and number of worst places for comparison and shown them with different colors. Clearly, the *Abstract Slicing* algorithm gives better results as compared to *APNSlicing* algorithm for almost every bench mark case study except *Dining Philosophers*. There is no reduction achieved by both algorithms for the *Dining Philosophers*. For the *Producer Consumer* and *Kanban*, *APNSlicing* algorithm also preforms worst whereas there is a significant reduction achieved by *Abstract Slicing* algorithm for both case studies.

Table 6.7: Applying *APNSlicing* algorithm

System	*T.States*	*Bst.Reduct*	*Avg.Reduct*	*Worst Places* (*no reduction*)
Complaint Handling	*2200*	*98.01%*	*40.54%*	*2*
House Construction	*10490*	*99.12%*	*95.12%*	*1*
Divide & Conquer	*13731*	*95.09%*	*14.22%*	*1*
Flexible Manufacturing System	*2895018*	*95.24%*	*80.21%*	*3*
Beverage Vending & Machine	*136*	*80.14%*	*02.15%*	*2*
Insurance Claim	*889*	*99.05%*	*24.52%*	*1*
A Customer support Production system	*471*	*99.01%*	*37.79%*	*zero*
Electronic Trade System	*260*	*97.56%*	*62.34%*	*2*
Daily Routine of 2 Employees & Boss	*80*	*93.75%*	*86.12%*	*zero*
Simple Protocol	*1861*	*95.91%*	*39.01%*	*1*
Producer Consumer	*372*	*0.00%*	*0.00%*	*5*
Kanban	*4600*	*0.00%*	*0.00%*	*16*
Dining Philosophers	*18*	*0.00%*	*0.00%*	*6*

When considering number of worst places for which *APNSlicing* does not produce any reduction it is also higher than *Abstract Slicing*. The *Abstract Slicing* algorithm gives reduction for every place of bench mark case studies except *Dining Philosophers*. 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*.

Table 6.8: Applying *Abstract Slicing* algorithm

System	*T.States*	*Bst.Reduct*	*Avg.Reduct*	*Worst Places* (*no reduction*)
Complaint Handling	*2200*	*98.01%*	*97.37%*	*zero*
House Construction	*10490*	*99.12%*	*98.72%*	*zero*
Divide & Conquer	*13731*	*99.09%*	*85.22%*	*zero*
Flexible Manufacturing System	*2895018*	*99.24%*	*95.21%*	*zero*
Beverage Vending & Machine	*136*	*80.14%*	*60.03%*	*zero*
Insurance Claim	*889*	*99.05%*	*94.52%*	*zero*
A Customer support Production system	*471*	*99.01%*	*37.79%*	*zero*
Electronic Trade System	*260*	*97.56%*	*77.34%*	*zero*
Daily Routine of 2 Employees & Boss	*80*	*93.75%*	*86.12%*	*zero*
Simple Protocol	*1861*	*95.91%*	*57.15%*	*zero*
Producer Consumer	*372*	*64.00%*	*36.00%*	*zero*
Kanban	*4600*	*89.05%*	*89.05%*	*zero*
Dining Philosophers	*18*	*0.00%*	*0.00%*	*6*

6.4 Applying slicing algorithm on practically relevant properties

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 is 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.

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:

$\phi_3 = \mathbf{AG}(NM \neq \emptyset)$, *where* "NM" represents a place not meeting.

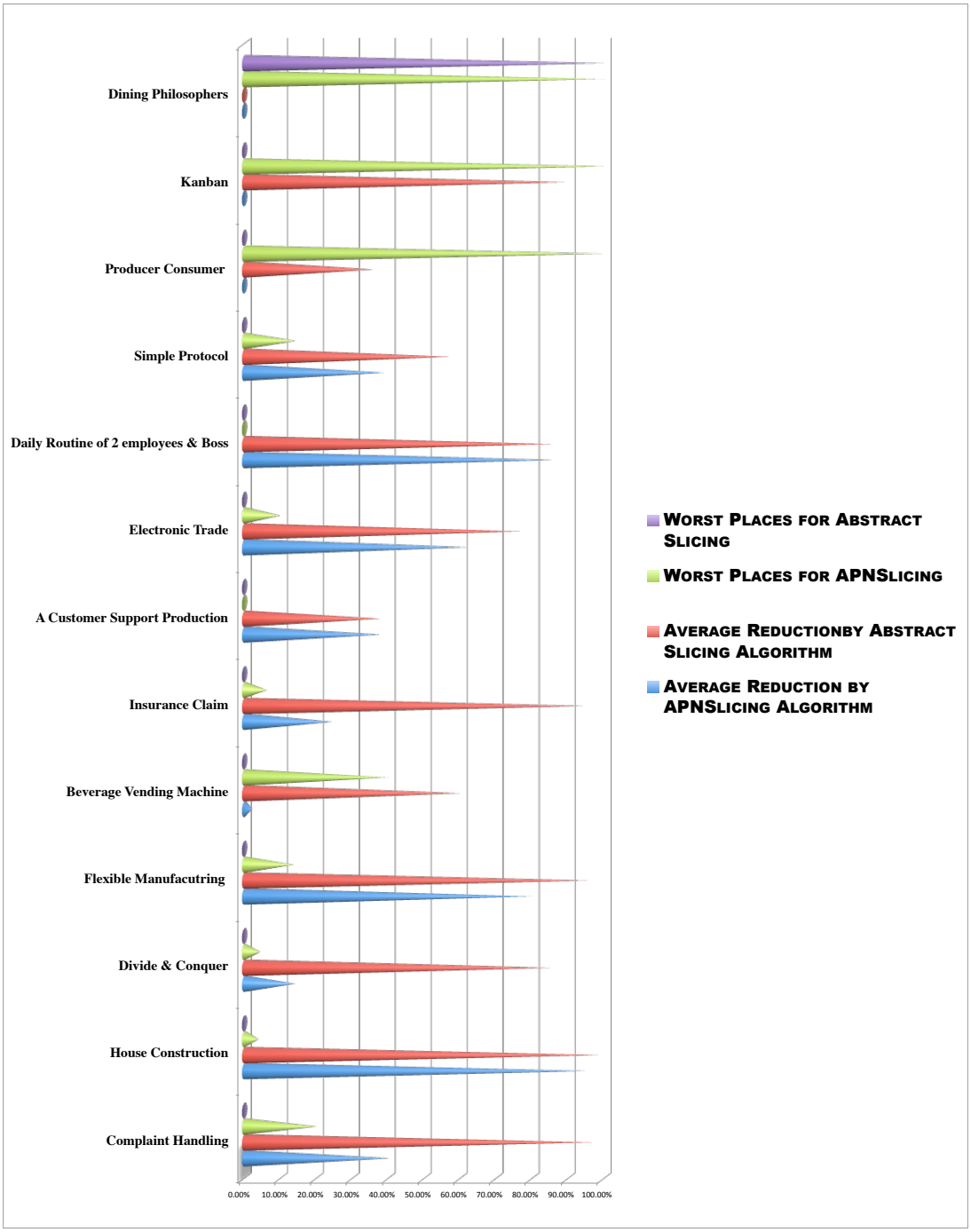


Figure 6.10: Comparison of *APNSlicing* and *Abstract Slicing* algorithms

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

$\phi_4 = \mathbf{AF}(|PackTorec| = |PackTosend|)$, *where* "PackTosend and PackTorec" represents places.

Another property could be *"All the recorded packets are less than five "*. Formally, we can specify the property:

$\phi_5 = \mathbf{AG}(token \in PackTorec \Rightarrow token < 5)$.

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

$\phi_6 = \mathbf{AG}(RecComp \Rightarrow \mathbf{AF}CompReg)$, *where* "RecComp" (resp. CompReg) means "place RecComp (resp. CompReg) 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:

$\phi_7 = \mathbf{AG}(|Requests| < 10)$.

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:

$\phi_8 = \mathbf{AG}(|Buffer| > 0)$.

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:

$\phi_9 = \mathbf{AG}(AC \Rightarrow \mathbf{AF}CS)$, *where* "AC" (resp. CS) means "place AC (resp. CS) is not empty".

Another property could be be to verify that *"Settled claims are always less than 10"*. Formally, we can specify the property:

$\phi_{10} = \mathbf{AG}(|CS| < 10)$.

For *Beverage Vending APN model* an interesting property could be to verify that: *"Machine has always drinks"*. Formally, we can specify the property:

$\phi_{11} = \mathbf{AG}(|Drinks| > 0)$.

For *Kanban APN model* an interesting property could be to verify that: *"Eventually first module is finished"*. Formally, we can specify the property:

$\phi_{12} = \mathbf{AF}(|P1| = 0)$.

For the *Dining Philosopher model* an interesting property could be to verify that: *"Eventually philosophers has left and right fork"*. Formally, we can specify the property:

$\phi_{13} = \mathbf{AF}(|HasL| > 0 \wedge |HasR| > 0)$, "HasL" (resp. HasR) are places in the net .

First of all, we applied *APNSlicing* to slice for the above mentioned properties and results are summarized in the Table.6.9.

Let us study the results summarized in the table shown in Table.6.9, the first column represents the system under observation whereas in the second column total number of states are given based on the initially marked places. The third column refers the property that we are looking for the verification. In the fourth column, places are given that are considered as criterion places, and for those places slices are generated. The fifth column represents the number of states that are reduced (in percentage) after applying *APNSlicing* algorithm.

Secondly, we applied *Abstract Slicing* algorithm for the above mentioned properties and results are summarized in the Table.6.10. Results are summarized similarly in the Table.6.10 as Table.6.9.

Table 6.9: Results with different properties concerning APN models by Applying *APNSlicing* algorithm

System	*Property*	*Tot.States*	*APNslicing*	*Reduction*
Car Crash	φ_1	*324*	*196*	*39.51%*
–	φ_2	*324*	*196*	*39.51%*
Daily Routine of 2 Employees & Boss	φ_3	*80*	*5*	*93.75%*
Simple Protocol	φ_4	*21*	*21*	*0.00%*
–	φ_5	*21*	*21*	*0.00%*
Complaint Handling	φ_6	*2200*	*679*	*69.14%*
A Customer support Production system	φ_7	*471*	*171*	*63.70%*
Producer Consumer	φ_8	*372*	*372*	*0.00%*
Insurance Claim	φ_9	*889*	*121*	*86.39%*
–	φ_{10}	*889*	*121*	*86.39%*
Beverage Vending Machine	φ_{11}	*136*	*136*	*0.00%*
Kanban	φ_{12}	*4600*	*4600*	*0.00%*
Dining Philosophers	φ_{13}	*18*	*18*	*0.00%*

Let us compare the results of both algorithms (i.e., *APNSlicing* and *Abstract Slicing* algorithms) with each other to see their performance in terms of state space reduction with respect to the above mentioned properties. Let us study the graph shown in the Fig.6.11, along the *y-axis* of the graph properties about different case studies are listed whereas along the *x-axis* number of states are given to compare the reduction impact. We took three parameters i.e., total number of states, number of states required to verify properties through *APNSlicing* and *Abstract Slicing* for comparison and shown them with different colors. Clearly, the *Abstract Slicing* algorithm gives better results as compared to *APNSlicing* algorithm for almost every bench mark case study except *Dining Philosophers*. There is no reduction achieved for both algorithms for ϕ_{13}. For the properties ϕ_8 and ϕ_{12}, *APNSlicing* algorithm also preforms worst whereas there is a significant reduction achieved by *Abstract Slicing* algorithm for both properties.

So far, we have applied only *APNSlicing* and *Abstract Slicing* algorithms. Now, we shall apply our *Liveness Slicing* algorithm. As discussed in the previous section, *Live-*

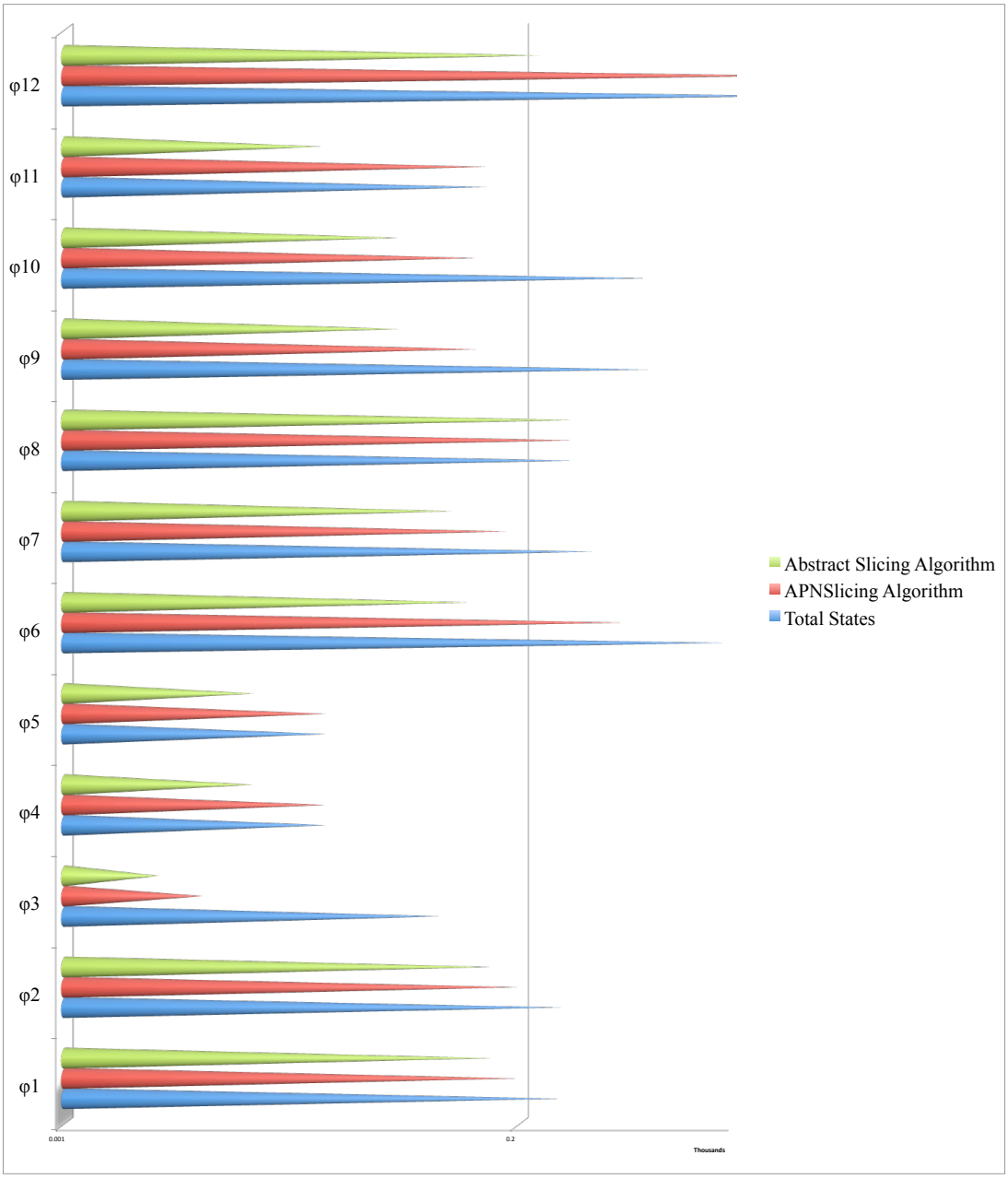


Figure 6.11: Comparison of *APNSlicing* and *Abstract Slicing* algorithms

Table 6.10: Results with different properties concerning to APN models by Applying *Abstract Slicing* algorithm

System	*Property*	*Tot.States*	*Abstract Slicing*	*Reduction*
Car Crash	φ_1	*324*	*144*	*55.56%*
–	φ_2	*324*	*144*	*55.56%*
Daily Routine of 2 Employees & Boss	φ_3	*80*	*3*	*96.25%*
Simple Protocol	φ_4	*21*	*9*	*57.14%*
–	φ_5	*21*	*9*	*57.14%*
Complaint Handling	φ_6	*2200*	*112*	*94.91%*
A Customer support Production system	φ_7	*471*	*91*	*80.68%*
Producer Consumer	φ_8	*372*	*372*	*0.00%*
Insurance Claim	φ_9	*889*	*49*	*94.48%*
–	φ_{10}	*889*	*49*	*94.48%*
Beverage Vending Machine	φ_{11}	*136*	*20*	*85.30%*
Kanban	φ_{12}	*4600*	*252*	*94.53%*
Dining Philosophers	φ_{13}	*18*	*18*	*0.00%*

ness Slicing algorithm is applicable to particular class of temporal formulas. Therefore, we shall take an example property from the above mentioned bench mark case studies and will apply *Liveness Slicing* algorithm together with the other two algorithms i.e., *APNSlicing* and *Abstract Slicing* algorithms.

For an *Insurance claim APN model*, if we are interested to verify: *"There exists a path for which claims are settled eventually"*. Formally, we can specify the property:

$\phi_{14} = \mathbf{EF}(|CompReg| > 0)$.

Let us compare the results obtained by applying *APNSlicing*, *Abstract Slicing* and *Liveness Slicing* algorithms) with each other to see their performance in terms of state space reduction with respect to ϕ_{14}. A comparison graph is shown in the Fig.6.12, along the *y-axis* of the graph ϕ_{14} is listed whereas along the *x-axis* number of states in percentage are given to compare the reduction impact. The graph shows number of

states required to verify ϕ_{14} by *APNSlicing*, *Abstract Slicing* and *Liveness Slicing* algorithms and total number of states. Clearly, the *Liveness Slicing* algorithm performs better than other algorithm for such particular temporal formulas.

Table 6.11: Results with different properties concerning to APN models by Applying *Liveness Slicing* algorithm

System	*Property*	*Tot.States*	*Liveness Slicing*	*Reduction*
Insurance Claim	φ_{14}	*889*	*25*	*98.19%*

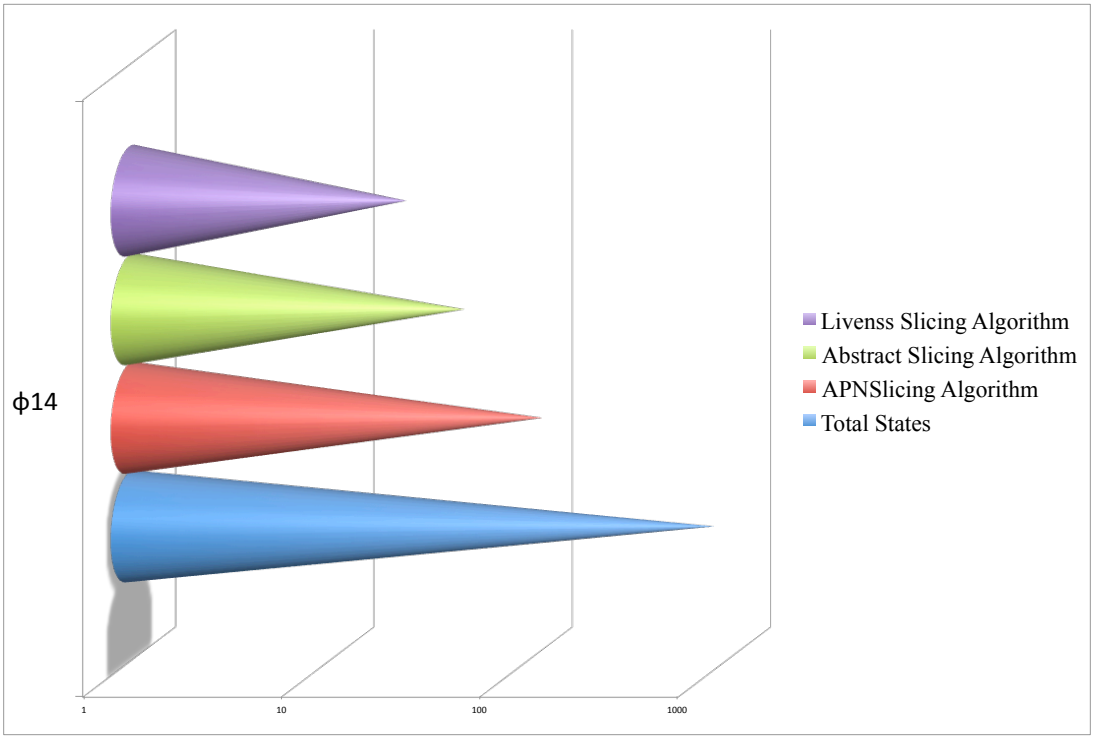


Figure 6.12: Comparison of *APNSlicing*, *Abstract Slicing* and *Liveness Slicing* algorithms

We can draw the following conclusions from the evaluation results:

- Reduction can vary with respect to the net structure and markings of the places. 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.

- The choice of the place can have an important influence on the reduction effects, as the basic idea of slicing is to start from the criterion place and iteratively include all the transitions contributing tokens on them together with their incoming places. The fewer transitions are attached to the criterion place, the more reduction is possible.

- *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* in its construction.

- For certain strongly connected nets slicing may produce a reduced number of states. For all the strongly connected nets, that contain *reading transitions* slicing can produce noteworthy reductions.

- It has been empirically proved that in general slicing produces best results for work-flow nets in [Rak12, Kha14]. Our experiments also prove that for work-flow nets abstract slicing produces better results.

- All the proposed slicing algorithms are linear time complex.

Chapter 7

SLAP$_n$: A tool for slicing Petri nets and Algebraic Petri nets

To understand the best is to work on its implementation.
— Jean-Marie Guayau

The first proposal of slicing Petri nets was presented in 1994 and still many research groups are working on new proposals for slicing Petri nets and their different classes. Despite the fact that it is has produced good research results, there is no tool support available publicly. This is a first effort to build a stand alone tool in the context of PN Slicing to the best of our knowledge. We have developed a small prototype implementation of the proposed slicing algorithms, called the SLAP$_n$.

One of the main objective of SLAP$_n$ is to show the practical usability of slicing on different classes of Petri nets. It mainly focuses on Algebraic Petri nets and low-level Petri nets. This tool has a small scope without great ambition, but it could be the basis for a much richer work, as SLAP$_n$ offers some important perspectives for merging it with the existing model checking tools (such as *AlPiNA*, *CPNTOOLS*, *TINA*, *TAPAAL*). It is important to note that SLAP$_n$ is not a model checker itself but it can be embedded into existing model checkers as a pre-processing step. There are two major types of slicing algorithms, which are *static & dynamic slicing* algorithms. The *static slicing* algorithms are designed to improve model checking whereas *dynamic slicing* are designed for the improvement of testing. In the first version of our tool, we implemented only *static slicing* algorithms, which are:

- **APNSlicing**
- **Abstract Slicing**
- **Safety Slicing**
- **Liveness Slicing**

It is important to note that we do not implement *dynamic slicing* algorithms and left their implementation as a future work.

Fig.7.1 gives an overview of the activities involved in SLAP$_n$ using Process Flowchart. First of all, user can draw unfolded APN model or PN model by using modeling tools provided in the tool. Our proposed slicing algorithms starts by taking APN/PN model and criterion places. These criterion places are extracted automatically by the tool from the temporal description of properties. Finally, user can choose slicing algorithm, she wish to apply and a sliced APN/PN model is produced. The sliced net is then used to generate the state space to verify given properties to existing model checkers.

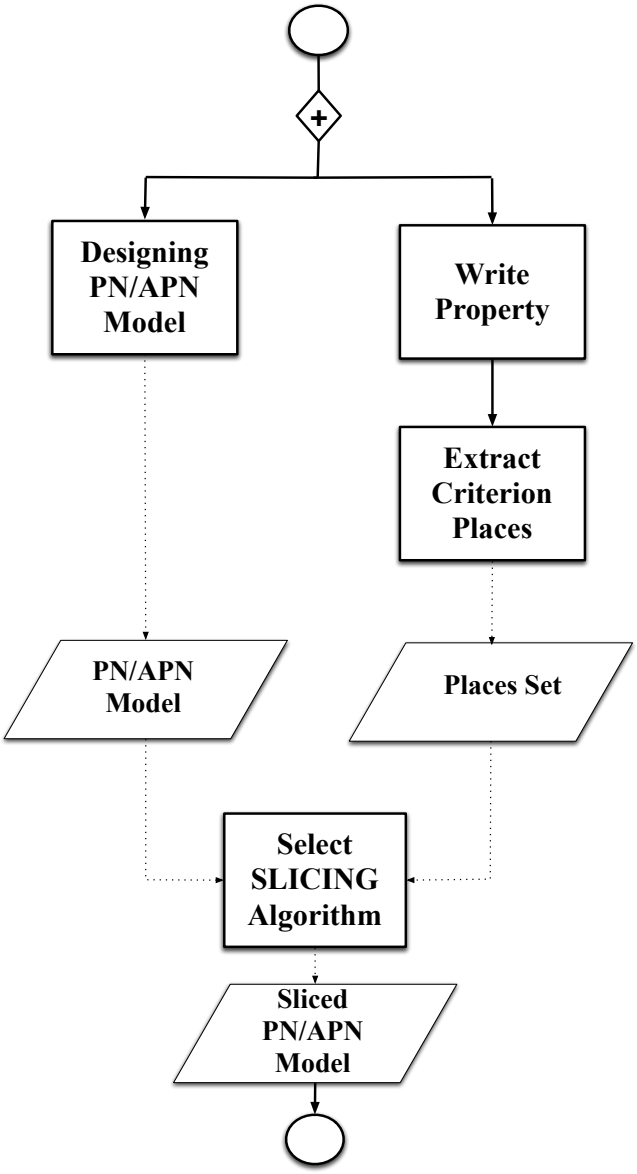


Figure 7.1: Expanded Process flow of SLAP$_n$

7.1 Overview

The SLAP$_n$ tool is an eclipse plugin that allows to create APN/PN models in a user-friendly interface, and apply different slicing algorithms to slice the created models [Ecla]. The meta model of SLAP$_n$ is shown in Fig.7.2. The SLAP$_n$ class contains slicing algorithms, temporal description of properties and Algebraic Petri nets classes. Different slicing algorithms such as *APNSlicing, AbstractSlicing LivenessSlicing, SafetySlicing and ConcernedSlicing* are extension to the slicing algorithms class.

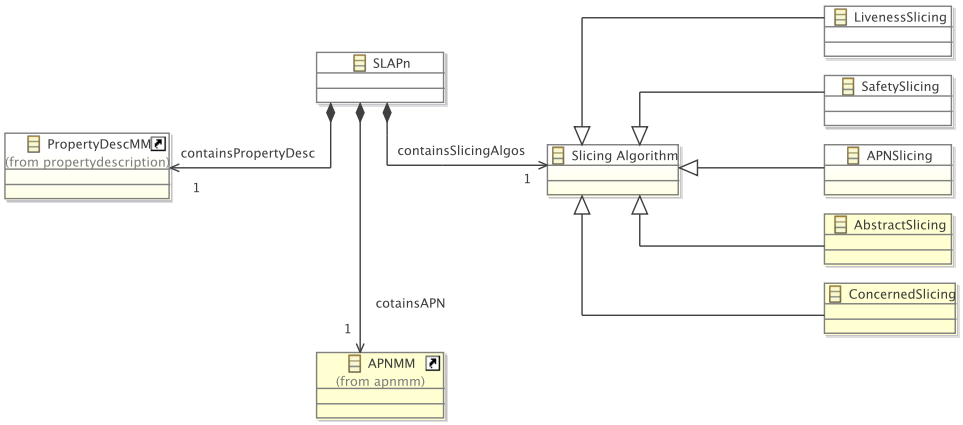


Figure 7.2: SLAP$_n$ Meta model

Being an Eclipse plugin, it shares many of the user interface mechanism with the Eclipse IDE. Fig. 7.3 describes the SLAP$_n$ perspective. It is composed of following areas:

- Editor provides a central place to create and modify the models.
- The model navigator presents the different files that compose a model. Their extension tell whether they are graphical models of unfolded APN/PN (.pnmm) or textual files with (.pnmm_diagram).
- The toolbar gives quick access to write temporal formula and to apply implemented slicing algorithms.

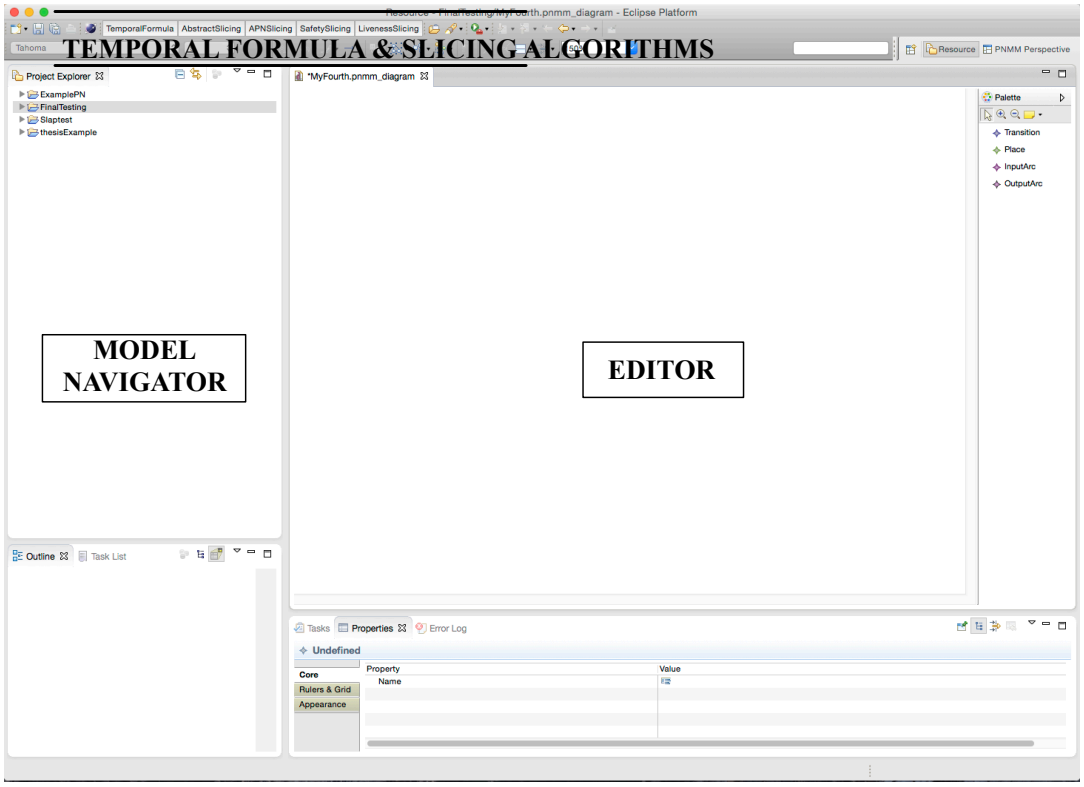


Figure 7.3: SLAP$_n$ main screen

From the architectural point of view, SLAP$_n$ consists of graphical editor and a slicing engine (i.e., consisting different implementations of proposed slicing algorithms) as shown in Fig.7.4. The Graphical User Interface (GUI) is developed with the graphical and textual tools from Eclipse Modeling Project namely Eclipse Modeling Framework (EMF) for the creation of metamodels of unfolded APN/PN, Graphical Modeling Framework (GMF) for the creation of the graphical elements of the user interface. The slicing engine that consists of extraction of criterion places and slicing algorithms has been completely implemented using the Java platform.

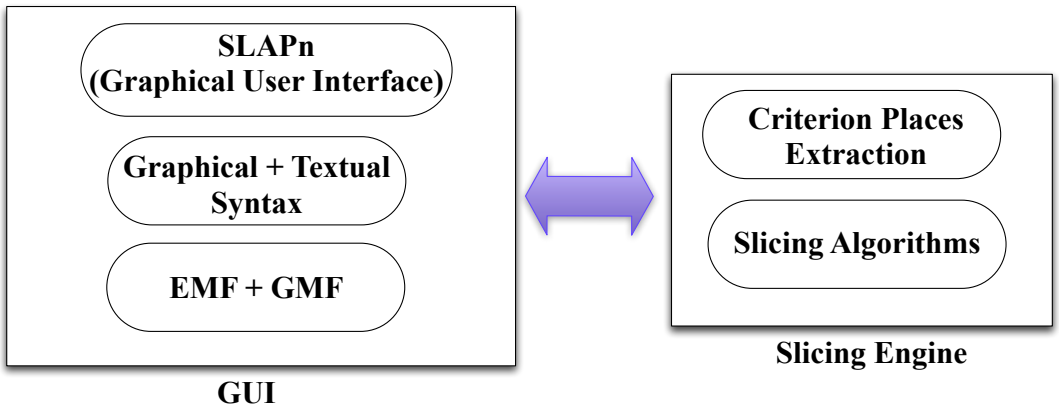


Figure 7.4: SLAP$_n$ architecture

Since 2002, EMF is a framework for modeling and code generation which is part of the Eclipse platform (http://www.eclipse.org/modeling/emf/) [Eclb]. EMF has three main components i.e., core (metametamodel, persistence, serialization, validation and model tracing); edit (model viewing and editing); and codegen (API generation for ECorebased models). The main advantage of using EMF is that the large palette of bundled tools makes the creation and manipulation of metamodels easy. Whereas GMF allows to create visual syntax for model editing, or XText to define textual editors with syntax checking, auto completion and other advanced features (http://www.eclipse.org/modeling/gmf/).

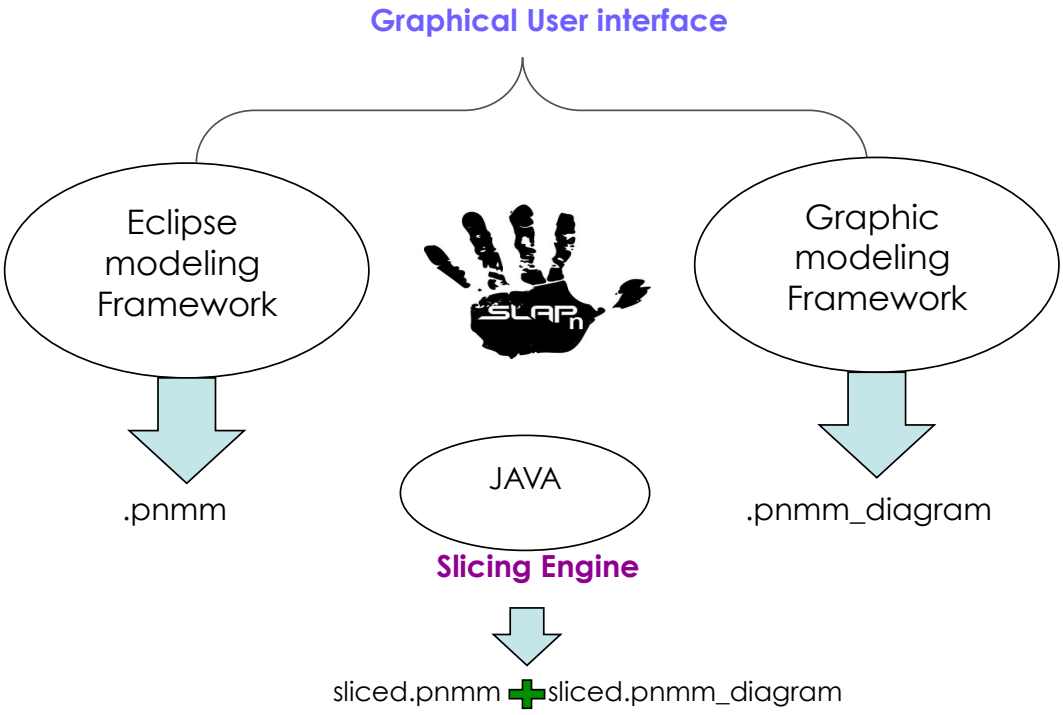


Figure 7.5: Slapn generated files

From the developer's point of view, there are two types of files that are created while creating a textual and graphical PN model i.e., .pnmm_*diagram* and .pnmm. The .pnmm files is stored in XML (EXtensible Markup Language) format. XML is a markup language much like HTML (Hyper Text Markup Language) and is used to describe the data. In XML tags are not predefined and designed to be self descriptive. Our implementation of slicing algorithms takes .pnmm file as an input. Below, we show .pnmm files for the example shown in the Fig.7.6. There are four different tags that are used to show the example PN model. The tags names `ContainsPlaces & ContainsTransitions` store information regarding the places (resp. transitions) in the PN model. The tags name `containsInputArcs & containsOutputArcs` store information about the input and output arcs in the PN model.

```
<?xml version="1.0" encoding="UTF-8"?>
<pnmm:PetriNets xmi:version="2.0" xmlns:xmi="http://www.omg.org/
    XMI" xmlns:pnmm="http://pnmm/1.0">
  <containsPlaces name="p1" numboftokens="2"/>
  <containsPlaces name="p0" numboftokens="2"/>
  <containsPlaces name="p2" numboftokens="1"/>
  <containsInputArcs weight="1" InputArcFroPlace="//
      @containsPlaces.1" InputArcToTransition="//
      @containsTransitions.0" placename="p0" transname="t1"/>
  <containsInputArcs weight="1" InputArcFroPlace="//
      @containsPlaces.0" InputArcToTransition="//
      @containsTransitions.1" placename="p1" transname="t1"/>
  <containsInputArcs weight="3" InputArcFroPlace="//
      @containsPlaces.2" InputArcToTransition="//
      @containsTransitions.2" placename="p2" transname="t3"/>
  <containsInputArcs weight="1" InputArcFroPlace="//
      @containsPlaces.2" InputArcToTransition="//
      @containsTransitions.3" placename="p2" transname="t2"/>
  <containsTransitions name="t0"/>
  <containsTransitions name="t1"/>
  <containsTransitions name="t3"/>
  <containsTransitions name="t2"/>
  <containsOutputArcs weight="1" OutputArcToPlace="//
      @containsPlaces.0" OutputArcFroTransition="//
      @containsTransitions.0" placename="p1" tranname="t0"/>
  <containsOutputArcs weight="2" OutputArcToPlace="//
      @containsPlaces.2" OutputArcFroTransition="//
      @containsTransitions.1" placename="p2" tranname="t1"/>
  <containsOutputArcs weight="3" OutputArcToPlace="//
      @containsPlaces.2" OutputArcFroTransition="//
      @containsTransitions.2" placename="p2" tranname="t3"/>
</pnmm:PetriNets>
```

7.2 Tasks in SLAP$_n$

- **Creating Models:** The first task is to create an APN/PN model. Four model elements are present inside palette i.e, Transition, Place, InputArc and outputArc.
- **Writing Temporal Formula:** Tool bar provides a button to write a temporal formula. Tool will automatically extract criterion places for which slices will be generated.
- **Apply Slicing Algorithm:** Finally, a user can select any algorithm given in the tool bar buttons. The slicing algorithm starts with the criterion places set and unfolded APN/PN model.

Let us draw a simple unfolded PN model and then apply slicing algorithms using SLAP$_n$. Fig.7.6 shows a screen shot of an example PN model that we draw using SLAP$_n$ model elements. Let us take an example temporal formula i.e., **AG**($|P1| \neq \emptyset$) and write it using the button in the tool bar as shown in Fig.7.7. Now let us first apply the *APNSlicing* algorithm and secondly apply *Abstract slicing* algorithm from the tool bar buttons, the resultant sliced models are shown in Fig.7.8, 7.9 respectively.

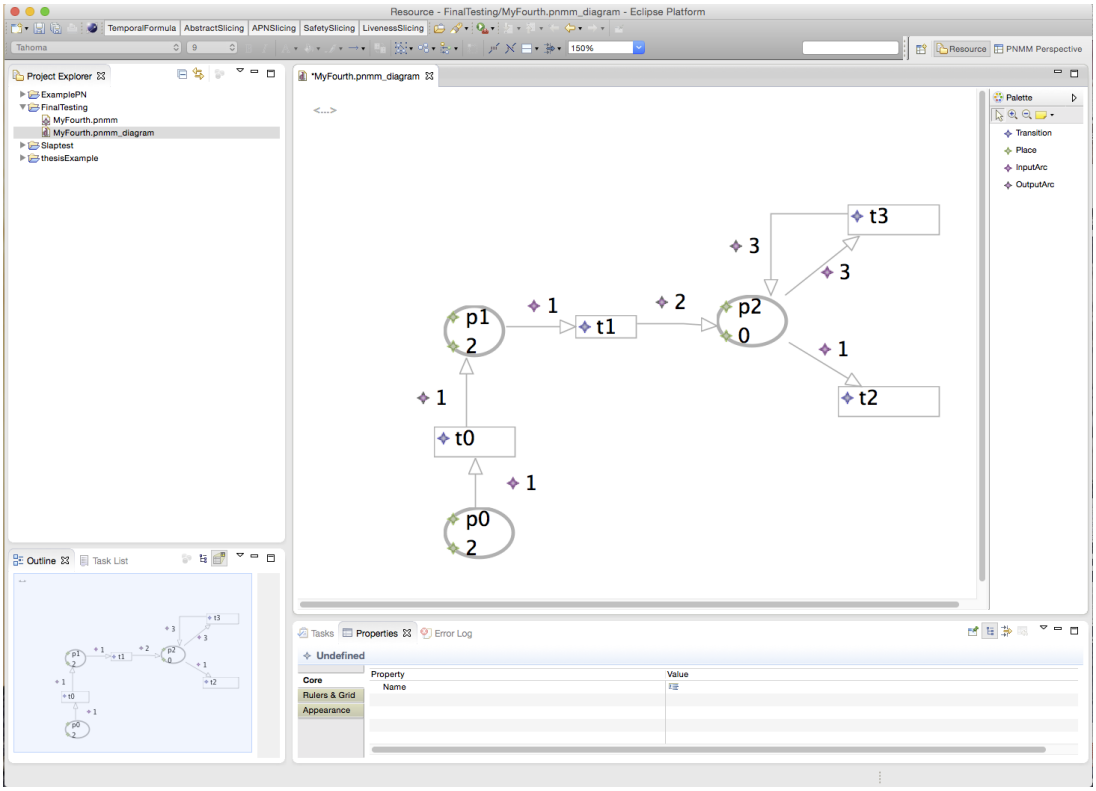


Figure 7.6: Drawing a APN model using SLAP$_n$ editor

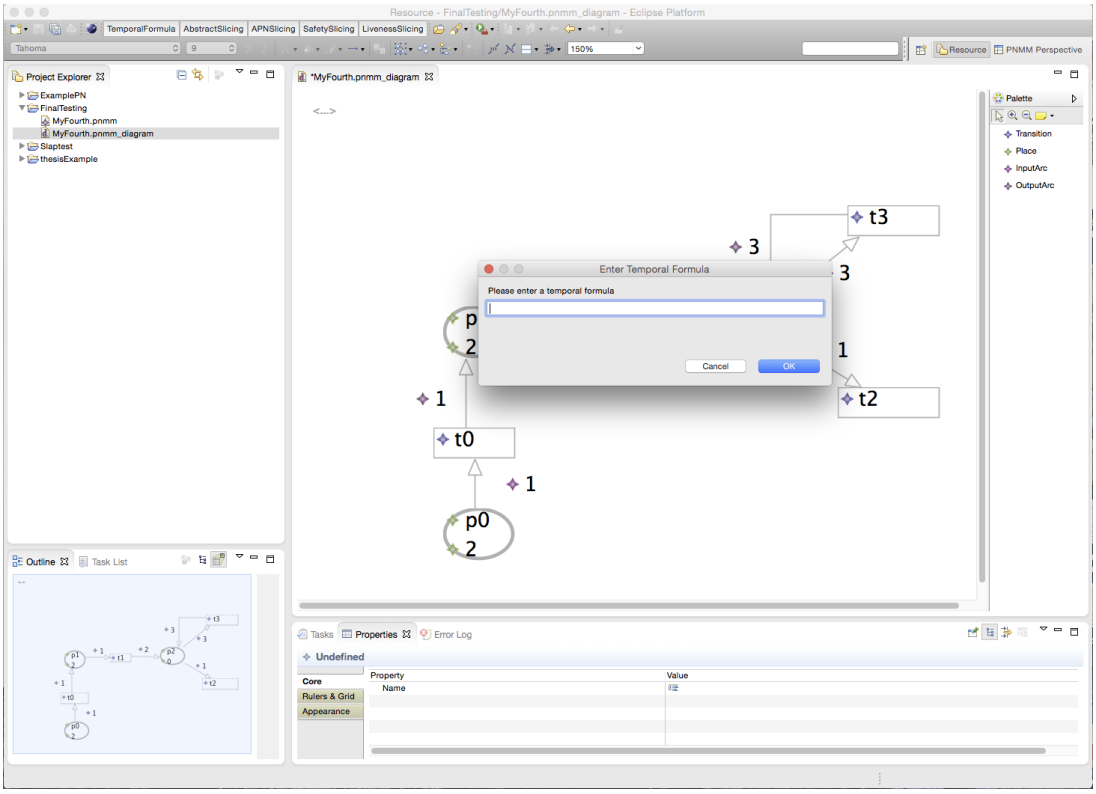


Figure 7.7: Writing temporal formula

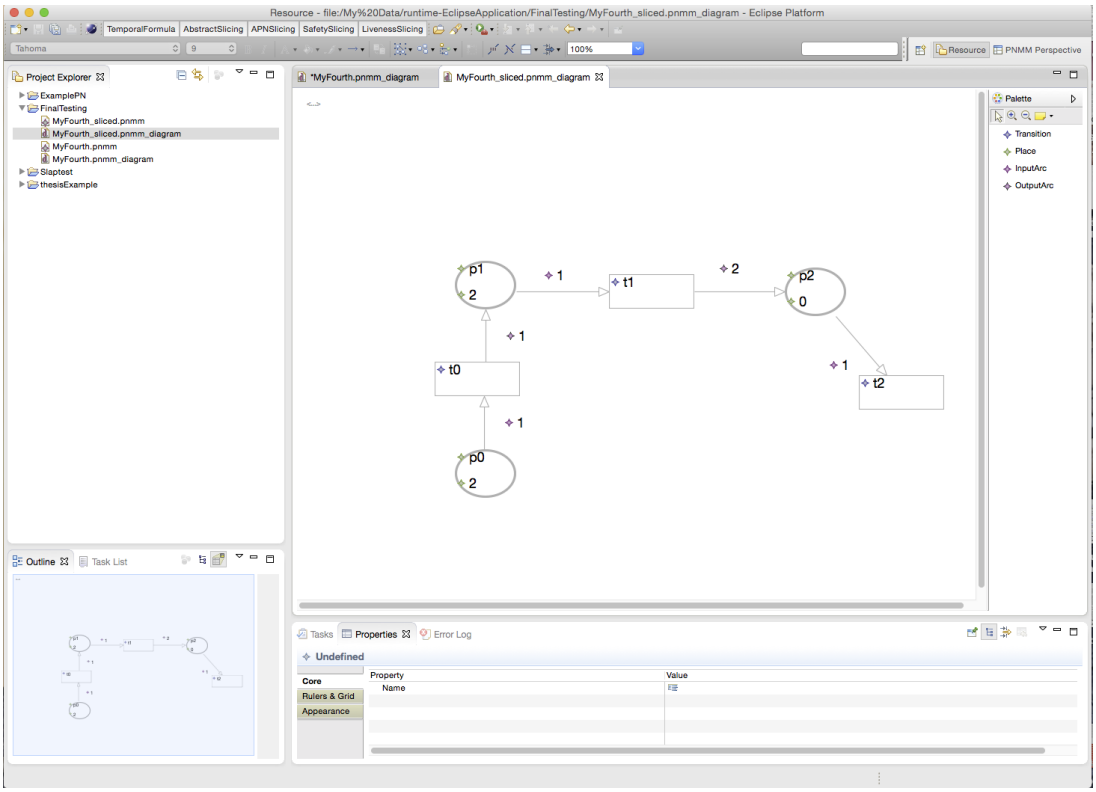


Figure 7.8: Sliced model by applying *APNSlicing* algorithm

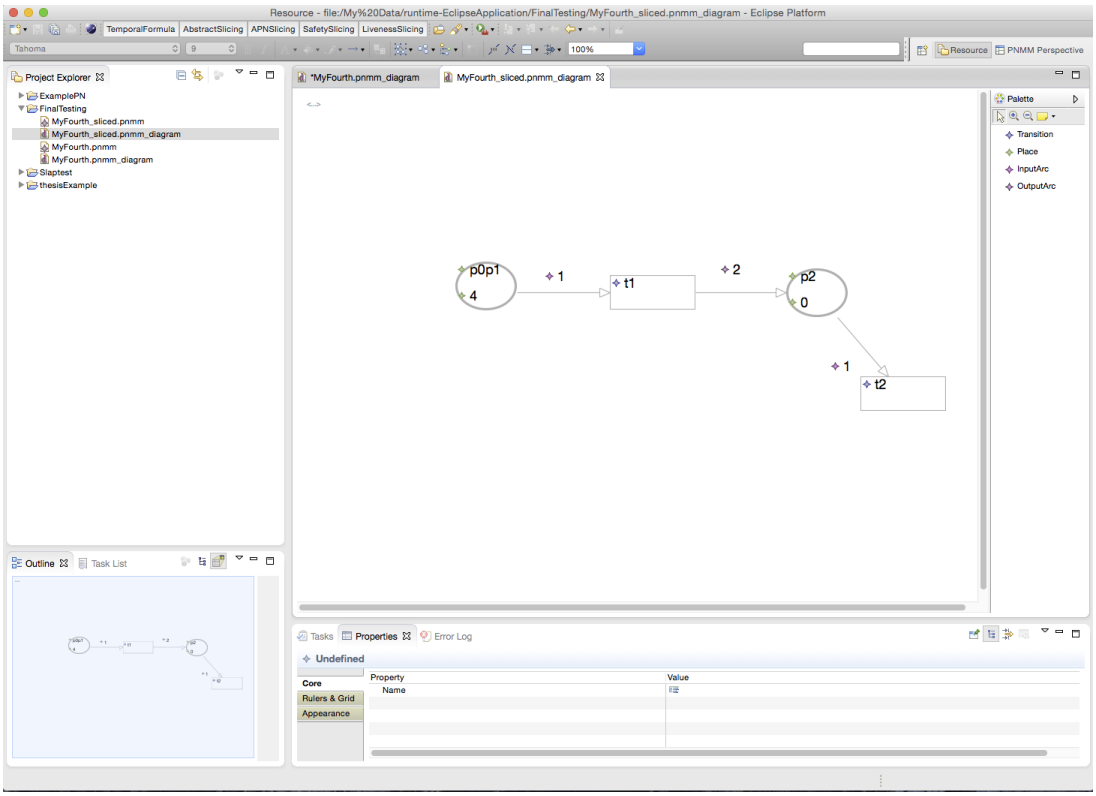


Figure 7.9: Sliced model by applying *Abstract slicing* algorithm

Chapter 8

Conclusion and Future work

To succeed, jump as quickly at opportunities as you do at conclusions.

— Benjamine Franklin

The main contribution of this thesis is to to improve verification and re-verification of concurrent and distributed system modeled in Petri nets. In this thesis, we presented property based model checking of Algebraic Petri nets, property based model checking of structurally evolving Algebraic Petri nets and SLAP$_n$ tool.

- **Property based model checking of Algebraic Petri nets:** a technique based on slicing has been successfully proposed to handle the state space explosion problem (see chapter 4). The presented approach proposes efficient slicing algorithms that can be applied to Petri nets and Algebraic Petri nets. The proposed algorithm can alleviate state space even for certain strongly connected nets. By construction, it is guaranteed that the state space of sliced net is at most as big as the original net. We showed that the slice allow verification and falsification if Algebraic Petri net is slice fair. The proposed slicing algorithms are linear time complex and significantly improves the model checking and testing of Algebraic Petri nets.
- **Property based model checking of structurally evolving Algebraic Petri nets:** a technique has been successfully proposed to pursue two goals; the first is to perform verification only on the parts that may affect the property the Algebraic Petri net model is analyzed for (see chapter 5). The second is to classify evolutions of Algebraic Petri nets to identify, which evolutions require verification. To give more flexibility to the user, we do not restrict the types of structural evolutions and the properties. Our results show that slicing is helpful to alleviate the state space explosion problem of Algebraic Petri nets model checking and the repeated model checking of structural evolutions of Algebraic Petri nets.

- **SLAP$_n$:** a tool for slicing Petri nets and Algebraic Petri nets has been successfully developed (see chapter 7). It is our believe that computer science research relies on proofs of concept and that it requires to implement the ideas to prove that a concept or algorithm is valuable. Therefore, all the algorithms and approaches have been implemented as open-source tools.

8.1 Future Work

The future work consists of following objectives:

- The first future work objective is to expand this dissertation work to build better slicing algorithms and their implementations to infer more about verification and re-verification of other interesting and expressive modeling formalisms. In order to build better slicing algorithms, one possible direction is to investigate how to use compositional reasoning method. The compositional method verifies each component of a system in the isolation and allows global properties to be inferred about the entire system. This method is not only better suited for improving verification, it can also be used to reason about the property satisfaction when an evolution happens to any component of the system. We believe combining compositional reasoning with slicing could provide an effective solution to the verification and re-verification.

- A tool name SLAP$_n$ (a tool for slicing Petri nets and Algebraic Petri nets) has been developed as a proof of concept for the proposed slicing algorithms. An important future work related to SLAP$_n$ is to integrate it with the existing model checking tools. In the present form of SLAP$_n$ only those slicing algorithms are implemented that are designed to improve model checking. Another future work consists of implementing slicing algorithms that are designed to improve testing. It is important to note that the SLAP$_n$ tool handles Petri nets and Algebraic Petri nets and it can be extended to other classes of Petri nets such as Colord Petri nets, Timed Petri nets.

- The second objective of future work is concerned to enhance the theory of preservation of properties. The aim is to develop a property preserving domain specific language for the evolving Petri nets based on the slicing and the classification of evolutions and properties proposed in this work.

Appendix A

Acronyms

Acronyms

AADT Algebraic Abstract Data Type. vii, 13, 82

ADT Algebraic Data Type. 7

APN Algebraic Petri Net. vii–ix, 6–8, 13, 16–18, 24, 43, 44, 47–51, 53–57, 60, 62–64, 66, 67, 69, 70, 72, 76, 80–82, 84, 85, 87, 88, 94, 96–101, 103–105, 107, 109, 120–122, 124

APNs Algebraic Petri Nets. iii, vii, 3, 5–9, 12, 13, 44, 46, 50, 53, 58, 62, 63, 76, 80, 82, 96, 103

BDD Binary Decision Diagram. 3, 24, 43

CPN Colord Petri Nets. 12

CTL Computation tree logic. 5

HLPN High Level Petri Net. 12, 15

HLPNs High-Level Petri nets. 3

HLRS High-level Replacement System. 7

ICT Information and Communication Technology. 1

LTL Linear temporal logic. 5

PN Petri nets. vii, xi, 2, 3, 5, 7, 8, 12, 13, 16, 25–29, 31, 34, 36, 37, 41, 103–105, 120–122, 124

PN Slicing Petri nets Slicing. 5, 6, 119

PN slicing Petri nets Slicing. 24–26, 29, 40, 46

PNs Petri nets. iii, 2–4, 6–9, 12, 13, 43, 44, 63, 80, 103

Pr/T-nets Predicate/Transition Nets. 12, 13

SLAP$_n$ A tool for Slicing Algebraic Petri nets. iii, iv, vii, ix, 8, 9, 105, 119–124

Appendix B

Algebraic Specifications for Car Crash Management System

In this appendix we present algebraic specifications for the carCrash case study.

B.0.1 Algebraic specifications for CCMS

```
1
2  Adt boolean
3      Sorts bool;
4      Generators
5          true : bool;
6          false : bool;
7      Operations
8          not : bool -> bool;
9          and : bool, bool -> bool;
10         or : bool, bool -> bool;
11         xor : bool, bool -> bool;
12         implies : bool, bool -> bool;
13
14     Axioms
15         //not
16         not(true) = false;
17         not(false) = true;
18
19         //and
20         and(true, $boolVar) = $boolVar;
21         and(false, $boolVar) = false;
22
23         //or
24         or(true, $boolVar) = true;
25         or(false, $boolVar) = $boolVar;
26
27         //xor
28         xor(true, $boolVar) = not($boolVar);
29         xor(false, $boolVar) = $boolVar;
30
31         //implies
32         implies(false, $boolVar) = true;
33         implies(true, $boolVar) = $boolVar;
34
35     Variables
36         boolVar : bool;
```

```
1
2 Adt capturedata
3
4     Sorts
5         capture;
6
7     Generators
8
9         Fire:capture;
10        Blockage:capture;
```

```
1 Adt observers
2     Sorts
3         obs;
4
5     Generators
6         YK:obs;
7         NG:obs;
```

```
1 import"observers.adt"
2 import"capturedata.adt"
3 Adt SuperObserver
4
5         Sorts
6             Sobs;
7         Generators
8
9             sobs: obs,capture->Sobs;
10        Operations
11
12            getsob:Sobs->obs;
13        Axioms
14
15            getsob(sobs($obs,$ct))=$obs;
16
17
18        Variables
19            obs:obs;
20            ct:capture;
```

```
1 import "capturedata.adt"
2
3 Adt RecordCrises
4
5     Sorts
6         crisesR;
7
8     Generators
9         cR: capture->crisesR;
10
11    Operations
12
13        getcapturetype:crisesR->capture;
14
15    Axioms
16        getcapturetype(cR($cd))= $cd;
17    Variables
18        cd:capture;
```

```
1 import"boolean.adt"
2 import"RecordCrises.adt"
3 import"capturedata.adt"
4     Adt
5         system
6         Sorts
```

```
7             sys;
8         Generators
9             system:crisesR ,bool->sys;
10        Operations
11            getcrisestype:sys ->capture;
12            getcrises:sys ->crisesR;
13        Axioms
14            getcrises(system($cr,$b))= $cr;
15            getcrisestype(system($cr,$b))= getcapturetype($cr);
16
17        Variables
18            cr:crisesR;
19            b:bool;
```

```
1 import "RecordCrises.adt"
2 import"system.adt"
3 import"SuperObserver.adt"
4 import"capturedata.adt"
5
6
7     Adt Executecrises
8
9         Sorts
10            crises;
11
12        Generators
13            assigncrises:Sobs ,sys->crises;
14        Operations
15            getcrisestype:crises ->capture;
16
17        Axioms
18            getcrisestype(assigncrises($sob,$sy))= getcrisestype($sy);
19
20        Variables
21            cr:crisesR;
22            sob:Sobs;
23            sy:sys;
```

```
1 import "Executecrises.adt"
2 import "boolean.adt"
3
4     Adt Report
5
6         Sorts
7             report;
8
9         Generators
10            rp: crises->report;
```

Appendix C

Java Program for APN Slicing Algorithm

Here is the java program for APN slicing algorithm given in the section [].

```
1  /*
2   This java program contains implementation of APNSlicing algorithm.
3   We use Document Object Model (DOM) standard whereas DOM. The Document Object Model
        provides APIs that  let you create, modify, delete, and rearrange nodes.
4  */
5  package com.slap.algorithms;
6  import java.io.File;
7  import java.util.ArrayList;
8  import java.util.List;
9  import javax.swing.JOptionPane;
10 import javax.xml.parsers.DocumentBuilder;
11 import javax.xml.parsers.DocumentBuilderFactory;
12 import javax.xml.transform.Transformer;
13 import javax.xml.transform.TransformerFactory;
14 import javax.xml.transform.dom.DOMSource;
15 import javax.xml.transform.stream.StreamResult;
16 import org.w3c.dom.Attr;
17 import org.w3c.dom.Document;
18 import org.w3c.dom.Element;
19 import org.w3c.dom.Node;
20 import org.w3c.dom.NodeList;
21
22 public class APNSlicing {
23
24     public static String inputPlace = null;
25     public static String variableplace = null;
26     public static String variabletrans = null;
27     public static String variableinputarc = null;
28     public static String variabletrans1 = null;
29     public static String variabletrans2 = null;
30     public static String existingPlace = null;
31     public static String variableswitch;
32
33     public static String transdel = null;
34
35     public static String[] placeArray = new String[10];
36     public static String[] inputarcArray = new String[10];
37     public static String[] outputarcArray = new String[0];
38
39     public static List<String> list = new ArrayList<String>();
40
41     public static List<String> listClone = new ArrayList<String>();
42
43     public static List<String> listinputarcslice = new ArrayList<String>();
```

```
44      public static List<String> listinputarc = new ArrayList<String>();
45      public static List<String> listoutputarc = new ArrayList<String>();
46      public static boolean bool = false;
47      public static boolean isTransitionExist = false;
48      public static String[] transArray = new String[10];
49      public static int i = 0;
50      public static int j;
51      public static int k = 0;
52
53      public static void main(String arg[]) {
54          // TODO input place to the algorithm
55          String variableplace = JOptionPane.showInputDialog("Enter Temporal Formula");
56          //String variableplace = "P1";
57          // System.out.println(variableplace);
58           variableswitch = existingPlace;
59
60          // System.out.println("Enter Place to Start with : ");
61          inputPlace = variableplace;
62          list.add("");
63
64          try {
65
66              // TODO input file i.e. Petri net
67              File fXmlFile = new File(
68                      "/My Data/runtime-EclipseApplication/Finaltesting/MyFirstPN.pnmm"
                            );
69              DocumentBuilderFactory dbFactory = DocumentBuilderFactory
70                      .newInstance();
71              DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
72              Document doc = dBuilder.parse(fXmlFile);
73              doc.getDocumentElement().normalize();
74
75              // System.out.println("Root element :" +
76              // doc.getDocumentElement().getNodeName());
77
78              //TODO existing XML nodes reading with their tag names
79              NodeList transitionList = doc.getElementsByTagName("containsTransitions")
                    ;
80              NodeList placeList = doc.getElementsByTagName("containsPlaces");
81              NodeList inputArcsList = doc.getElementsByTagName("containsInputArcs");
82              NodeList outputArcsList = doc.getElementsByTagName("containsOutputArcs");
83
84
85              //TODO Output XML file tag names i.e. Temporary names. Later on, it will
                    be converted to original tag names
86              NodeList placeListSl = doc.getElementsByTagName("containsPlacesSl");
87              NodeList transitionListSl = doc.getElementsByTagName("
                    containsTransitionsSl");
88              NodeList inputArcsListSl = doc.getElementsByTagName("containsInputArcsSl"
                    );
89              NodeList outputArcsListSl = doc.getElementsByTagName("
                    containsOutputArcsSl");
90
91              //TODO New/output XML file root element
92              Element root = doc.getDocumentElement();
93
94              for (int t = 0; t < placeList.getLength(); t++) {
95                  //TODO Checking either this input place existed or not
96                  if (!(existingPlace + "").equalsIgnoreCase(inputPlace)) {
97
98                      //TODO Finding place in existing places/xml file and creating new
                            place for output xml file
99                      findAndCreatePlace(doc, placeList, root);
100
101                     //TODO Find and Create input arcs
102                     findAndCreateInputArcs(doc, inputArcsList, root);
103
104                     //TODO find and create output arc
```

```
105                     findAndCreateOutputArcs(doc, outputArcsList, root);
106
107                     //TODO Exploring input arc, that it is not coming from existing
                            place
108                     findPlaceNameToCreate(inputArcsList);
109
110                 }//end out most if
111
112             }//end outer most for loop
113
114
115             //TODO comparing with already generated transitions and adding places to
                    a new list
116             inputArcsOfReadingTransitions(inputArcsListSl);
117
118             //TODO comparing with already generated transitions and adding places to
                    a new list
119             outputArcsOfReadingTransitions(outputArcsListSl);
120
121
122             comparingInputAndOutputPlaces();
123
124             if (k < 1 & k != 2) {
125
126                 getValuesOfReadingTransitions(inputArcsListSl);
127
128                 compareInputAndOutputArcsOfReadingTransitions(inputArcsListSl,
                        outputArcsListSl);
129
130                 removeReadingTransitions(transitionListSl);
131
132                 removingInputArcTransitions(inputArcsListSl);
133
134                 removingOutputArcTransition(outputArcsListSl);
135
136             }
137
138             if (inputPlace != null) {
139
140                 removeOldTagNames(placeList);
141                 removeOldTagNames(transitionList);
142                 removeOldTagNames(inputArcsList);
143                 removeOldTagNames(outputArcsList);
144
145
146                 renameTags(doc, placeListSl,"containPlaces");
147                 renameTags(doc, transitionListSl,"containsTransitions");
148                 renameTags(doc, inputArcsListSl,"containsInputArcs");
149                 renameTags(doc, outputArcsListSl,"containsOutputArcs");
150
151             }
152
153             TransformerFactory transformerFactory = TransformerFactory
154                     .newInstance();
155             Transformer transformer = transformerFactory.newTransformer();
156             DOMSource source = new DOMSource(doc);
157             StreamResult result = new StreamResult(
158                     new File(
159                             "/My Data/runtime-EclipseApplication/Finaltesting/
                                    MyFirstPN1.pnmm"));
160             transformer.transform(source, result);
161         }
162
163         catch (Exception e) {
164             e.printStackTrace();
165         }
166
167     }
```

```
168
169     /**
170      * Exploring input arc, that it is not coming from existing place
171      * @param inputArcsList
172      */
173     public static void findPlaceNameToCreate(NodeList inputArcsList) {
174         for (int temp11 = 0; temp11 < inputArcsList.getLength(); temp11++) {
175
176             Node InpArcList1 = inputArcsList.item(temp11);
177             Element eElement5 = (Element) InpArcList1;
178
179             String valueOfTransname1 = eElement5.getAttribute("transname");
180             String valueOfPlacename1 = eElement5.getAttribute("placename");
181             // System.out.println("this is old place  "+inputPlace+"");
182             // System.out.println("this is place from list "+valueOfPlacename+"");
183             // System.out.println("variable switch value "+variableswitch);
184
185             //TODO Exploring input arc, that it is not coming from existing place
186             if (valueOfTransname1.equalsIgnoreCase(variabletrans) & valueOfPlacename1
                    != inputPlace) {
187                 // System.out.println("variable switch value"
188                 // +variableswitch );
189                 System.out.println("values of different places"+ inputPlace);
190
191                 // System.out.println("the value"+variabletrans);
192                 existingPlace = inputPlace;
193                 inputPlace = valueOfPlacename1;
194
195                 System.out.println("my grand value " + existingPlace);
196
197                 // break;
198             }
199
200         }//end for loop Exploring input arc, that it is not coming from existing
                place
201     }
202
203     /**
204      * Taking created place as an input and creating output arcs list
205      * @param doc
206      * @param outputArcsList
207      * @param root
208      */
209     public static void findAndCreateOutputArcs(Document doc,
210             NodeList outputArcsList, Element root) {
211         for (int temp3 = 0; temp3 < outputArcsList.getLength(); temp3++) {
212
213             Node outpArcList = outputArcsList.item(temp3);
214             Element outputArcElement = (Element) outpArcList;
215
216             // getting values of attributes from the already existed
217             // Input arc node
218             String valueOfWeightOfOutputArc = outputArcElement.getAttribute("weight")
                    ;
219             String valueOfOutputArcToPlace = outputArcElement.getAttribute("
                    OutputArcToPlace");
220             String valueOfOutputArcFroTransition = outputArcElement.getAttribute("
                    OutputArcFroTransition");
221             String valueOfPlacenameout = outputArcElement.getAttribute("placename");
222             String valueOfTransnameout = outputArcElement.getAttribute("tranname");
223
224             //TODO Checking if output arc exist for this input place
225             if (outputArcElement.getAttribute("placename").equalsIgnoreCase(
                    inputPlace + "")) {
226
227                 Element SliceOutputArc = doc.createElement("containsOutputArcsSl");
228                 root.appendChild(SliceOutputArc);
229                 // System.out.println("created outputarc");
```

```
230
231                 if (valueOfWeightOfOutputArc != "") {
232                     Attr placeWeightout = doc.createAttribute("weight");
233                     SliceOutputArc.setAttributeNode(placeWeightout);
234                     placeWeightout.setValue(valueOfWeightOfOutputArc);
235                 }
236                 Attr InputArcFroPlaceout = doc.createAttribute("OutputArcToPlace");
237                 SliceOutputArc.setAttributeNode(InputArcFroPlaceout);
238                 InputArcFroPlaceout.setValue(valueOfOutputArcToPlace);
239
240                 Attr InputArcToTransitionout = doc.createAttribute("
                        OutputArcFroTransition");
241                 SliceOutputArc.setAttributeNode(InputArcToTransitionout);
242                 InputArcToTransitionout.setValue(valueOfOutputArcFroTransition);
243
244                 Attr placenameout = doc.createAttribute("placename");
245                 SliceOutputArc.setAttributeNode(placenameout);
246                 placenameout.setValue(valueOfPlacenameout);
247
248                 Attr transnameout = doc.createAttribute("tranname");
249                 SliceOutputArc.setAttributeNode(transnameout);
250                 transnameout.setValue(valueOfTransnameout);
251
252                 listoutputarc.add(valueOfTransnameout);
253                 variabletrans = valueOfTransnameout;
254
255                 // variabletrans2= valueOfTransnameout;
256                 // s = valueOfPlacenameout;
257                 System.out.println("output arc is created "+ variabletrans);
258                 inputPlace = valueOfPlacenameout;
259
260                 //TODO checking if out put transition already created or not
261                 createOutputTransitions(doc, root);
262
263             }//end if Checking if output arc exist for this input place
264
265         }// end for loop find and create output arc
266     }
267
268     /**
269      * Creating output transitions
270      * @param doc
271      * @param root
272      */
273     public static void createOutputTransitions(Document doc, Element root) {
274         for (i = 0; i < list.size(); i++) {
275
276             if (list.get(i).equals(variabletrans)) {
277                 System.out.println("found value in output tranistion "+ list.get(i));
278                 /*
279                  * for (String string : list) {
280                  * if(!string.contains(variabletrans)){
281                  */
282                 isTransitionExist = true;
283                 break;
284
285             } else {
286                 isTransitionExist = false;
287             }
288         }//end for loop checking if out put transition already created or not
289
290         //TODO if transition does not exist, then create and add to to the list
291         if (isTransitionExist != true) {
292             Element Slicetrans = doc.createElement("containsTransitionsSl");
293             root.appendChild(Slicetrans);
294
295             Attr vtransname = doc.createAttribute("name");
296             Slicetrans.setAttributeNode(vtransname);
```

```
297             vtransname.setValue(variabletrans + "");
298             // System.out.println("the newly create transition by tag name "
299             // +eElement11.getAttribute("name"));
300             if (list.get(0) == ("")) {
301                 list.remove(0);
302             }
303             list.add(variabletrans);
304 
305             System.out.println("transition is created "+ variabletrans);
306             // break;
307             // b=false;
308         }//end if transition does not exist, then create and add to to the list
309     }
310 
311     /**
312      * Taking create place as an input and creating input arcs list
313      * @param doc
314      * @param inputArcsList
315      * @param root
316      */
317     public static void findAndCreateInputArcs(Document doc,
318             NodeList inputArcsList, Element root) {
319         for (int temp1 = 0; temp1 < inputArcsList.getLength(); temp1++) {
320 
321             Node InpArcList = inputArcsList.item(temp1);
322             Element inputArcElement = (Element) InpArcList;
323 
324 
325             String valueOfWeightOfInputArc = inputArcElement.getAttribute("weight");
326             String valueOfInputArcFroPlace = inputArcElement.getAttribute("
                    InputArcFroPlace");
327             String valueOfInputArcToTransition = inputArcElement.getAttribute("
                    InputArcToTransition");
328             String valueOfPlacename = inputArcElement.getAttribute("placename");
329             String valueOfTransname = inputArcElement.getAttribute("transname");
330 
331             //TODO navigating input arc element from input xml file
332             if (inputArcElement.getAttribute("placename").equalsIgnoreCase(inputPlace
                     + "")) {
333 
334                 Element SliceInputArc = doc.createElement("containsInputArcsSl");
335                 root.appendChild(SliceInputArc);
336 
337                 if (valueOfWeightOfInputArc != "") {
338                     Attr placeWeight = doc.createAttribute("weight");
339                     SliceInputArc.setAttributeNode(placeWeight);
340                     placeWeight.setValue(valueOfWeightOfInputArc);
341                 }
342 
343                 Attr InputArcFroPlace = doc.createAttribute("InputArcFroPlace");
344                 SliceInputArc.setAttributeNode(InputArcFroPlace);
345                 InputArcFroPlace.setValue(valueOfInputArcFroPlace);
346 
347                 Attr InputArcToTransition = doc.createAttribute("InputArcToTransition
                        ");
348                 SliceInputArc.setAttributeNode(InputArcToTransition);
349                 InputArcToTransition.setValue(valueOfInputArcToTransition);
350 
351                 Attr placename = doc.createAttribute("placename");
352                 SliceInputArc.setAttributeNode(placename);
353                 placename.setValue(valueOfPlacename);
354 
355                 Attr transname = doc.createAttribute("transname");
356                 SliceInputArc.setAttributeNode(transname);
357                 transname.setValue(valueOfTransname);
358 
359                 String valuetransname = SliceInputArc.getAttribute("transname");
360                 variableswitch = valueOfPlacename;
```

```
361
362                 // listinputarc.add(valuetransname);
363                 variabletrans = valuetransname;
364
365                 System.out.println("input arc created "+ variabletrans);
366
367                 //TODO Checking either this transition is already created or not
368                 createInputTransitions(doc, root);
369
370             }// end if navigating input arc element from input xml file
371
372         }// end for loop Find and Create input arcs
373     }
374
375     /**
376      * Creating input transitions
377      * @param doc
378      * @param root
379      */
380     public static void createInputTransitions(Document doc, Element root) {
381         for (i = 0; i < list.size(); i++) {
382             // String n = list.get(i);
383
384             System.out.println("value of boolean variable "+ isTransitionExist);
385             if (list.get(i).equals(variabletrans)) {
386                 System.out.println("found value input transitions " + list.get(i));
387
388                 /*
389                  * for (String string : list) {
390                  * if(!string.contains(variabletrans)){
391                  */
392                 isTransitionExist = true;
393                 break;
394
395                 // System.out.println(variableplace);
396
397             } else {
398                 isTransitionExist = false;
399             }
400
401         }// end forloop Checking either this transition is already created or not
402
403         //TODO if transition does not exists in list, then add it to the transition
                list
404         if (isTransitionExist != true) {
405             Element Slicetrans = doc.createElement("containsTransitionsSl");
406             root.appendChild(Slicetrans);
407
408             Attr vtransname = doc.createAttribute("name");
409             Slicetrans.setAttributeNode(vtransname);
410             vtransname.setValue(variabletrans + "");
411             // System.out.println("the newly create transition by tag name "
412             // +eElement11.getAttribute("name"));
413             if (list.get(0) == ("")) {
414                 list.remove(0);
415             }
416
417             list.add(variabletrans);
418             System.out.println("transition is created "+ variabletrans);
419             // b= false;
420
421             // break;
422         }//end if transition does not exists in list, then add it to the transition
                list
423     }
424
425     /**
426      * Taking criterion places as an input and comparing input with existing places
```

```
            in original xml file
427         * @param doc original xml document
428         * @param placeList new places will be added in this list
429         * @param root xml root element
430         */
431        public static void findAndCreatePlace(Document doc, NodeList placeList,
432                Element root) {
433            String variableplace;
434            for (int temp = 0; temp < placeList.getLength(); temp++) {
435
436                Node pList = placeList.item(temp);
437                Element eElement = (Element) pList;
438                String nameOfPlace = eElement.getAttribute("name");
439
440                if (nameOfPlace.equals(inputPlace)) {
441
442                    String tok = eElement.getAttribute("numboftokens");
443
444                    Element SlicePlac = doc.createElement("containsPlacesSl");
445                    root.appendChild(SlicePlac);
446
447                    //TODO setting up the attributes of newly created element
448                    Attr placeName = doc.createAttribute("name");
449                    SlicePlac.setAttributeNode(placeName);
450                    placeName.setValue(inputPlace + "");
451
452
453                    //TODO Checking if noOfTokens attribute exists, then create a new
                            element and add it to the newly created element
454                    if (tok != "") {
455                        Attr placetokens = doc.createAttribute("numboftokens");
456                        SlicePlac.setAttributeNode(placetokens);
457                        placetokens.setValue(tok + "");
458                    }
459
460                    // String valueofFirstPlace = eElement.getAttribute("name");
461                    // System.out.println("valueofFirstPlace"+valueofFirstPlace);
462                    variableplace = nameOfPlace;
463                    existingPlace = nameOfPlace;
464
465                    System.out.println("the new place is created "+ inputPlace);
466                }//end if
467
468            }//end forloop finding place and creating new place in output xml file
469        }
470
471        /**
472         * inputArcsOfReadingTransitions
473         * @param inputArcsListSl
474         */
475        public static void inputArcsOfReadingTransitions(NodeList inputArcsListSl) {
476            for (int temp11 = 0; temp11 < inputArcsListSl.getLength(); temp11++) {
477
478                Node InpArcList1 = inputArcsListSl.item(temp11);
479
480                Element eElement5 = (Element) InpArcList1;
481
482                String valueOfTransname = eElement5.getAttribute("transname");
483                String valueOfPlacename = eElement5.getAttribute("placename");
484
485                for (i = 0; i < list.size(); i++) {
486
487                    if (list.get(i).equals(valueOfTransname)) {
488
489                        listinputarc.add(valueOfPlacename);
490
491                        System.out.println("chcking the values of input arcs "
492                                + valueOfPlacename);
```

```
493
494                 }
495             }
496
497         }//end for loop comparing with already generated transitions and adding
                places to a new list
498     }
499
500     /**
501      * outputArcsOfReadingTransitions
502      * @param outputArcsListSl
503      */
504     public static void outputArcsOfReadingTransitions(NodeList outputArcsListSl) {
505         for (int temp11 = 0; temp11 < outputArcsListSl.getLength(); temp11++) {
506
507             Node outputArcList1 = outputArcsListSl.item(temp11);
508
509             Element eElement5 = (Element) outputArcList1;
510
511             String valueOfTransname = eElement5.getAttribute("tranname");
512             String valueOfPlacename = eElement5.getAttribute("placename");
513
514             // System.out.println("my value at stat "+listinputarc.get(i));
515             // System.out.println("value of places outputarc"+valueOfTransname);
516
517             for (i = 0; i < list.size(); i++) {
518
519                 // System.out.println("my value at stat 2 "+valueOfTransname);
520                 System.out.println("my value at stat 3 " + valueOfPlacename);
521
522                 if (list.get(i).equals(valueOfTransname)) {
523                     for (i = 0; i < outputArcsListSl.getLength(); i++) {
524                         System.out.println("my value at stat "+ listinputarc.get(i));
525
526                         if (listinputarc.get(i).equals(valueOfPlacename)) {
527
528                             listinputarcslice.add(valueOfPlacename);
529                             variableinputarc = valueOfPlacename;
530                             System.out.println("assigned value of place name out "+
                                    valueOfPlacename);
531                         }
532
533                         else {
534
535                             System.out.println("else condtion updated");
536                             k = 2;
537
538                         }
539
540                     }
541
542                 }
543             }
544
545         }
546     }
547
548     /**
549      * Comparing input places and output places of reading transitions.
550      */
551     public static void comparingInputAndOutputPlaces() {
552         for (i = 0; i < listinputarcslice.size(); i++) {
553             if (!listinputarcslice.get(i).equals(variableinputarc)) {
554                 k = 1;
555             }
556         }
557     }
558
```

```
559     /**
560      * Getting values of reading transitions
561      * @param inputArcsListSl
562      */
563     public static void getValuesOfReadingTransitions(NodeList inputArcsListSl) {
564         for (int temp11 = 0; temp11 < inputArcsListSl.getLength(); temp11++) { //
                start
565             // of
566             // second
567             // loop
568 
569             Node InpArcList1 = inputArcsListSl.item(temp11);
570 
571             Element eElement5 = (Element) InpArcList1;
572 
573             String valueOfTransname = eElement5
574                     .getAttribute("transname");
575             String valueOfPlacename = eElement5
576                     .getAttribute("placename");
577 
578             if (valueOfPlacename.equals(variableinputarc)) {
579                 System.out
580                         .println("the values of places inside sliced version "
581                                 + valueOfTransname);
582 
583                 transdel = valueOfTransname;
584                 // InpurArcList1.getParentNode().removeChild(node);
585                 System.out.println(" i shall delete as well "
586                         + inputArcsListSl.getLength());
587                 // System.out.println("value of parent node "+eElement5.getNodeName()
                        );
588                 // eElement5.removeChild(InpArcList1) ;
589 
590                 // InpArcList1.getParentNode().removeChild(eElement5.getAttributeNode
                        (valueOfPlacename));
591 
592                 // eElement5.getParentNode().removeChild(eElement5.getAttributes()));
593 
594                 // System.out.println(" the attribute name of transition "+eElement5.
                        hasAttribute("transname"));
595 
596             }
597 
598         }
599     }
600 
601     /**
602      * compareInputAndOutputArcsOfReadingTransitions
603      * @param inputArcsListSl
604      * @param outputArcsListSl
605      */
606     public static void compareInputAndOutputArcsOfReadingTransitions(
607             NodeList inputArcsListSl, NodeList outputArcsListSl) {
608         for (int temp11 = 0; temp11 < outputArcsListSl.getLength(); temp11++) { //
                start
609             // of
610             // second
611             // loop
612 
613             Node InpArcList1 = outputArcsListSl.item(temp11);
614 
615             Element eElement5 = (Element) InpArcList1;
616 
617             String valueOfTransname = eElement5
618                     .getAttribute("tranname");
619             String valueOfPlacename = eElement5
620                     .getAttribute("placename");
621 
```

```
622             System.out.println("the values of transitions output "
623                     + valueOfTransname);
624             // System.out.println("the values of transitions output2 "+transdel);
625             // System.out.println("the values of transitions output3 "+
                    variableinputarc);
626             if (valueOfPlacename.equals(variableinputarc)
627                     & transdel.equals(valueOfTransname)) {
628                 // System.out.println("the values of places ouput  inside sliced
                        version "+valueOfTransname);
629
630                 transdel = valueOfTransname;
631                 // InpurArcList1.getParentNode().removeChild(node);
632                 System.out.println(" i shall delete as well "
633                         + inputArcsListSl.getLength());
634                 // System.out.println("value of parent node "+eElement5.getNodeName()
                        );
635                 // eElement5.removeChild(InpArcList1) ;
636
637                 // InpArcList1.getParentNode().removeChild(eElement5.getAttributeNode
                        (valueOfPlacename));
638
639                 // eElement5.getParentNode().removeChild(eElement5.getAttributes()));
640
641                 // System.out.println(" the attribute name of transition "+eElement5.
                        hasAttribute("transname"));
642
643             }
644
645         }
646     }
647
648     /**
649      * Removing reading transitions
650      * @param transitionListSl
651      */
652     public static void removeReadingTransitions(NodeList transitionListSl) {
653         for (int tt = 0; tt < transitionListSl.getLength(); tt++) {
654
655             Node transListSl = transitionListSl.item(tt);
656
657             Element eElement12 = (Element) transListSl;
658
659             if (eElement12.getAttribute("name").equals(transdel)) {
660                 transListSl.getParentNode().removeChild(eElement12);
661                 // System.out.println("i shall detel");
662             }
663
664         }
665     }
666
667     /**
668      * Removing input arcs
669      * @param inputArcsListSl
670      */
671     public static void removingInputArcTransitions(NodeList inputArcsListSl) {
672         for (int temp11 = 0; temp11 < inputArcsListSl.getLength(); temp11++) { //
                start
673             // of
674             // second
675             // loop
676
677             Node InpArcList1 = inputArcsListSl.item(temp11);
678
679             Element eElement5 = (Element) InpArcList1;
680
681             String valueOfTransname = eElement5
682                     .getAttribute("transname");
683             String valueOfPlacename = eElement5
```

```
684                         .getAttribute("placename");
685
686                 if (valueOfPlacename.equals(variableinputarc)
687                         & valueOfTransname.equals(transdel)) {
688
689                     InpArcList1.getParentNode().removeChild(eElement5);
690                 }
691
692             }
693         }
694
695         /**
696          * Comparing input and output arcs and removing reading transitions
697          * @param outputArcsListSl
698          */
699         public static void removingOutputArcTransition(NodeList outputArcsListSl) {
700             for (int temp11 = 0; temp11 < outputArcsListSl.getLength(); temp11++) { //
                 start
701                 // of
702                 // second
703                 // loop
704
705                 Node InpArcList1 = outputArcsListSl.item(temp11);
706
707                 Element eElement5 = (Element) InpArcList1;
708
709                 String valueOfTransname = eElement5
710                         .getAttribute("tranname");
711                 String valueOfPlacename = eElement5
712                         .getAttribute("placename");
713
714                 if (valueOfPlacename.equals(variableinputarc)
715                         & transdel.equals(valueOfTransname)) {
716
717                     InpArcList1.getParentNode().removeChild(eElement5);
718                 }
719
720             }
721         }
722
723         /**
724          * It removes already existing tag names
725          * @param oldList list of tag names in original xml file
726          */
727         public static void removeOldTagNames(NodeList oldList) {
728             while (oldList.getLength() > 0) {
729                 Node node = oldList.item(0);
730                 node.getParentNode().removeChild(node);
731             }
732         }
733
734         /**
735          * It renames newly created tags to older ones
736          * @param doc original document
737          * @param oldListSl original tag list
738          * @param newTag to replace with
739          */
740         public static void renameTags(Document doc, NodeList oldListSl, String newTag) {
741             String newTagName = newTag;
742             for (int i = 0; i < oldListSl.getLength(); i++) {
743                 doc.renameNode(oldListSl.item(i), null, newTagName);
744             }
745         }
746
747 }
```

Bibliography

[Bab91] R.G. Babb. Issues in the specification and design of parallel programs. In *Software Specification and Design, 1991., Proceedings of the Sixth International Workshop on*, pages 75–82, Oct 1991.

[BBF+10] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and software verification: model-checking techniques and tools*. Springer Publishing Company, Incorporated, 2010.

[BBG01] O. Biberstein, Didier Buchs, and Nicolas Guelfi. Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism. In *Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets.*, pages 73–130, 2001.

[BCC98] Sergey Berezin, Sérgio Vale Aguiar Campos, and Edmund M. Clarke. Compositional reasoning in model checking. In *Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, COMPOS'97, pages 81–102, London, UK, UK, 1998. Springer-Verlag.

[BCC07] Francesco Basile, Ciro Carbone, and Pasquale Chiacchio. Simulation and analysis of discrete-event control systems based on petri nets using {PNetLab}. *Control Engineering Practice*, 15(2):241 – 259, 2007.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. *Symbolic model checking without BDDs*. Springer, 1999.

[BCG95] Girish Bhat, Rance Cleaveland, and Orna Grumberg. Efficient on-the-fly model checking for ctl. In *Logic in Computer Science, 1995. LICS'95. Proceedings., Tenth Annual IEEE Symposium on*, pages 388–397. IEEE, 1995.

[BCM+90] J. R. Burch, E.M. Clarke, K. L. McMillan, D.L. 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 e*, pages 428–439, 1990.

[BG96] David W Binkley and Keith Brian Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[BHMR10] Didier Buchs, Steve Hostettler, Alexis Marechal, and Matteo Risoldi. Alpina: A symbolic model checker. In Johan Lilius and Wojciech Penczek, editors, *Applications and Theory of Petri Nets*, volume 6128 of *Lecture Notes in Computer Science*, pages 287–296. Springer Berlin Heidelberg, 2010.

[Bin98] David Binkley. The application of program slicing to regression testing. *Information and software technology*, 40(11):583–594, 1998.

[BJS09] Joakim Byg, KennethYrke Jørgensen, and Jiří Srba. Tapaal: Editor, simulator and verifier of timed-arc petri nets. In Zhiming Liu and AndersP. Ravn, editors, *Automated Technology for Verification and Analysis*, volume 5799 of *Lecture Notes in Computer Science*, pages 84–89. Springer Berlin Heidelberg, 2009.

[BV06] Bernard Berthomieu and Francois Vernadat. Time petri nets analysis with tina. In *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*, pages 123–124. IEEE, 2006.

[CE82] EdmundM. Clarke and E.Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. 131:52–71, 1982.

[CES86] 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.

[CR94] Juei Chang and Debra J. Richardson. Static and dynamic specification slicing. In *In Proceedings of the Fourth Irvine Software Symposium*, 1994.

[Ecla] Eclipse. Eclipse Foundation. Eclipse platform. http://www.eclipse.org/.

[Eclb] Eclipse. Eclipse Modeling Project. http://www.eclipse.org/modeling/.

[EH86] E. Allen Emerson and Joseph Y. Halpern. “sometimes” and “not never” revisited: On branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, January 1986.

[Er97] Sibylle Peuk Er. Invariant property preserving extensions of elementary petri nets. Technical report, Technische Universitat Berlin, 1997.

[Erl00] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, May 2000.

[Gen87] H. J. Genrich. Predicate/transition nets. In *Advances in Petri Nets 1986, Part I on Petri Nets: Central Models and Their Properties*, pages 207–247, London, UK, UK, 1987. Springer-Verlag.

[GL79] Hartmann J. Genrich and Kurt Lautenbach. The analysis of distributed systems by means of predicate ? transition-nets. In *Semantics of Concurrent Computation*, pages 123–147, 1979.

[GL91] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16, 1991.

[GP93] Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In Costas Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 438–449. Springer Berlin Heidelberg, 1993.

[HNSY94] Thomas A Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and computation*, 111(2):193–244, 1994.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[Hol97] C.M. Holloway. Why engineers should consider formal methods. In *Digital Avionics Systems Conference, 1997. 16th DASC., AIAA/IEEE*, volume 1, pages 1.3–16–22 vol.1, Oct 1997.

[Jen81] Kurt Jensen. Coloured petri nets and the invariant-method. *Theoretical computer science*, 14(3):317–336, 1981.

[Jen87] Kurt 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.

[JKW07] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. In *INTERNATIONAL JOURNAL ON SOFTWARE TOOLS FOR TECHNOLOGY TRANSFER*, page 2007, 2007.

[KG14a] Yasir Imtiaz Khan and Nicolas Guelfi. Slapn: A tool for slicing algebraic petri nets. *APN*, 2(3):P1, 2014.

[KG14b] Yasir Imtiaz Khan and Nicolas Guelfi. Slicing high-level petri nets. In *International Workshop on Petri Nets and Software Engineering (PNSE'14)*, page 20, 2014.

[KGM10] Jörg Kienzle, Nicolas Guelfi, and Sadaf Mustafiz. Crisis management systems: A case study for aspect-oriented modeling. In Shmuel Katz, Mira Mezini, and Jörg Kienzle, editors, *Transactions on Aspect-Oriented Software Development VII*, volume 6210 of *Lecture Notes in Computer Science*, pages 1–22. Springer Berlin Heidelberg, 2010.

[Kha12] Yasir Imtiaz Khan. A formal approach for engineering resilient car crash management system. Technical Report TR-LASSY-12-05, University of Luxembourg, 2012.

[Kha13a] Yasir Imtiaz Khan. Optimizing verification of structurally evolving algebraic petri nets. In V. Kharchenko A. Gorbenko, A. Romanovsky, editor, *Software Engineering for Resilient Systems*, volume 8166 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013.

[Kha13b] Yasir Imtiaz Khan. Optmizing algebraic petri net model checking by slicing. Technical Report TR-LASSY-13-02, University of Luxembourg, 2013.

[Kha14] Yasir Imtiaz Khan. Slicing high-level petri nets. Technical Report TR-LASSY-14-03, University of Luxembourg, 2014.

[KL88] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[KR12] Yasir Imtiaz Khan and Matteo Risoldi. Language enrichment for resilient mde. In *Proceedings of the 4th international conference on Software Engineering for Resilient Systems*, SERENE'12, pages 76–90, Berlin, Heidelberg, 2012. Springer-Verlag.

[KR13] Yasir Imtiaz Khan and Matteo Risoldi. Optimizing algebraic petri net model checking by slicing. *International Workshop on Modeling and Business Environments (ModBE'13, associated with Petri Nets'13)*, 2013.

[Kri63] Saul A. Kripke. Semantical analysis of modal logic i normal modal propositional calculi. *Mathematical Logic Quarterly*, 9(5-6):67–96, 1963.

[KV01] Orna Kupferman and Moshe Y Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[Lam83] Leslie Lamport. What good is temporal logic. *Information processing*, 83:657–668, 1983.

[Lap05] J.-C. Laprie. Resilience for the scalability of dependability. In *Network Computing and Applications, Fourth IEEE International Symposium on*, pages 5–6, July 2005.

[LB03] C. Larman and V.R. Basili. Iterative and incremental developments. a brief history. *Computer*, 36(6):47–56, 2003.

[LKCK00] 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.

[LLV12] Moussa Amrani Qin Zhang Levi Lucio, Eugene Syriani and Hans Vangheluwe. Invariant preservation in iterative modeling. *Proceedings of the ME 2012 workshop*, 2012.

[LOS+08] 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, December 2008.

[McM92] Kenneth Lauchlin McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.

[Mä02] Marko Mäkelä. Maria: Modular reachability analyser for algebraic system nets, 2002.

[Pel94] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of the 6th International Conference on Computer Aided Verification*, CAV '94, pages 377–390, London, UK, UK, 1994. Springer-Verlag.

[Pet] Petrinets. Petri Nets Model Checkers Database. https://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html/.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Hamburg, 1962.

[PGE98] J. Padberg, M. Gajewsky, and C. Ermel. Rule-based refinement of high-level nets preserving safety properties. In *Fundamental approaches to Software Engineering*, pages 22123–8. Springer Verlag, 1998.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, Oct 1977.

[Rak08] Astrid 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.

[Rak11] Astrid Rakow. *Slicing and Reduction Techniques for Model Checking Petri Nets*. PhD thesis, University of Oldenburg, 2011.

[Rak12] Astrid Rakow. Safety slicing petri nets. In Serge Haddad and Lucia Pomello, editors, *Application and Theory of Petri Nets*, volume 7347 of *Lecture Notes in Computer Science*, pages 268–287. Springer Berlin Heidelberg, 2012.

[Rei91] Wolfgang Reisig. Petri nets and algebraic specifications. *Theor. Comput. Sci.*, 80(1):1–34, 1991.

[RWL+03] Anne Vinter Ratzer, Lisa Wells, Henry Michael Lassen, Mads Laursen, Jacob Frank Qvortrup, Martin Stig Stissing, Michael Westergaard, Søren Christensen, and Kurt Jensen. Cpn tools for editing, simulating, and analysing coloured petri nets. In *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets*, ICATPN'03, pages 450–462, Berlin, Heidelberg, 2003. Springer-Verlag.

[Sch94] Karsten Schmidt. T–invariants of algebraic petri nets. *Informatik– Bericht*, 1994.

[Sim03] L. Simoncini. Amsd: a dependability roadmap for the information society in europe. In *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*, pages 153–154, Oct 2003.

[Som06] Ian Sommerville. *Software Engineering: (Update) (8th Edition) (International Computer Science Series)*. Addison Wesley, June 2006.

[Tip95] F. Tip. A survey of program slicing techniques. *JOURNAL OF PROGRAMMING LANGUAGES*, 3:121–189, 1995.

[Val91] Antti Valmari. A stubborn attack on state explosion. In *Proceedings of the 2Nd International Workshop on Computer Aided Verification*, CAV '90, pages 156–165, London, UK, UK, 1991. Springer-Verlag.

[Val98] Antti 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.

[WCZX13] Yu Wangyang, Yan Chungang, Ding Zhijun, and Fang Xianwen. Extended and improved slicing technologies for petri nets. *High Technology Letters*, 19(1), 2013.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[XQZ+05] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, March 2005.