Optimizing Verification of Structurally Evolving Algebraic Petri Nets

Yasir Imtiaz Khan

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

Abstract. System models are subject to evolve during the development life cycle, along which an initial version goes through a series of evolutions, generally aimed at progressively reaching all the requested qualities (completeness, correctness etc.). Among the existing development methodologies the iterative and incremental one has been proved to be efficient for system development but lacks of support for an adequate verification process. When considering Algebraic Petri nets (APNs) for modeling and model checking for verification, all the proofs must be redone after each iteration which is impractical both in terms of time and space. In this work, we introduce an Algebraic Petri net slicing technique that optimizes the model checking of static or structurally evolving APN models. Furthermore, our approach is proposing a classification of evolutions dedicated to the improvement of model checking.

Keywords: Evolution, Model checking, Slicing.

1 Introduction

In the field of information and communication technology systems (ICST), software evolution is of utmost importance. Existing systems continue to evolve as they are never complete. The complexity of the system grows with the evolution and needs better solutions for its management. There are several keywords which speak about the evolution such as change, adaptation, variation, modification, transformation etc. For the development of evolving systems, iterative refinements and incremental developments are adapted often due to rapid validation of the developed features at finer granularity [8, 11]. 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 correctly. 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 developments is that there is no guarantee that after each iteration of the model, it still satisfies the previous properties. A naive solution is to repeat verification after every iteration, which is very expensive in terms of time and space. A verification technique such as model checking is commonly used for the analysis of such system. 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.

In this work, we propose a solution to optimize the verification of evolving systems by re-using, adapting and refining state of the art techniques. Our proposal pursues two main goals. The first is to perform verification only on those parts that may affect the property the model is analyzed for. The second is to classify the evolutions to identify which evolutions require 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 would be sufficient. A slicing based solution has been proposed for a property specific verification in the context of Algebraic Petri nets (APNs) [9]. We extend the previous work to optimize the verification of structural evolutions to APNs.

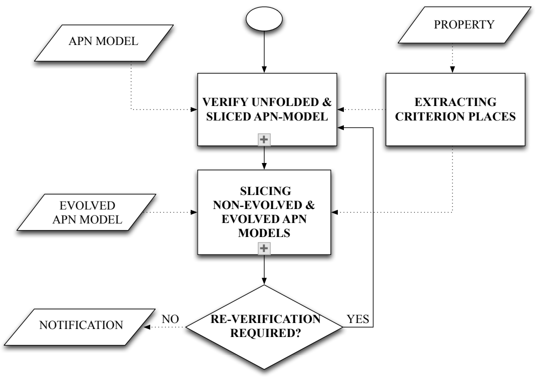
APN MODEL
PROPERTY
VERIFY UNFOLDED & SLICED APN-MODEL
EXTRACTING CRITERION PLACES
EVOLVED APN MODEL
SLICING NON-EVOLVED & EVOLVED APN MODELS
NOTIFICATION
NO
RE-VERIFICATION REQUIRED?
YES

Fig. 1. Process Flowchart optimizing verification of structurally evolving APN models

Fig.1, gives an overview using Process Flowchart of the proposed approach for slicing based verification of structurally evolving APNs . At first, verification is performed on the sliced unfolded APN model by taking a property into an account. Secondly, we build the slices for evolved and non-evolved APN models without unfolding them. Instead of verifying the full evolved APN model, we explore if the evolution has an impact on the property satisfaction. We perform verification for only those evolutions that impacts the property satisfaction. It is important to note that only the sliced evolved APN would be used to perform the verification. The process can be iterated as per APN evolutions.

The rest of the work is structured as follows: in the section 2, we give the formal definition of Algebraic Petri nets (APNs). The section 3 and 4, introduce all the steps of slicing based verification of structurally evolving APNs shown in the Fig.1. Details about the underlying theory and techniques are given for each activity of the process. In the section 5, we discuss related work and a comparison with the existing approaches. In section 6, we draw the conclusions and discuss future work concerning to the proposed work.

2 Formal Definition of Algebraic Petri Nets (APNs)

In this section, we give a basic formal definition of Algebraic Petri nets (APNs). Informally, Petri nets places hold resources (also known as tokens) and transitions are linked to places by input and output arcs, which can be weighted. Usually, a Petri net has a graphical concrete syntax consisting of circles for places, boxes for transitions and arrows to connect the two. The semantics of a Petri net involves the non-deterministic firing of transitions in the net. Firing a transition means consuming tokens from the set of places linked to the input arcs of the transition and producing tokens into the set of places linked to the output arcs of the transition. Various evolutions of Petri nets have been created, among others Algebraic Petri nets, that raise the level of abstraction of Petri nets by using complex structured data [19].

Definition 1. *A marked Algebraic Petri Net* $APN = < SPEC, P, T, F, asg, cond, \lambda, m_0 >$ *consist of*

∘ *an algebraic specification SPEC =* (Σ, E)*,*

∘ *P and T are finite and disjoint sets, called places and transitions, resp.,*

∘ $F \subseteq (P \times T) \cup (T \times P)$*, the elements of which are called arcs,*

∘ *a sort assignment* $asg : P \to S$*,*

∘ *a function,* $cond : T \to \mathcal{P}_{fin}(\Sigma - equation)$*, 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* $T_{OP,asg(p)}$*,*

∘ *an initial marking* m_0 *assigning a finite multiset over* $T_{OP,asg(p)}$ *to every place p.*

We refer the interested reader to [9] for the symbols and detailed algebraic specifications used in the formal definition of APNs for our work.

3 Unfolding, Slicing and Verifying APNs

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 concerns by the property. These 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.

Further refinement to the slicing constructions can be done by distinguishing between *reading* and *non-reading transitions.* The conception of *reading and non-reading transitions* is introduced in [18] for the first time. Informally, *reading transitions* are not supposed to change the marking of a place. On the other hand *non-reading transitions* are supposed to change the markings of a place. To identify a transition to be a *reading or non-reading* in a standard Petri nets, we compare the weights attached over the incoming and outgoing arcs from transition to place and place to transition. Excluding *reading transitions* and including only *non-reading transitions* significantly reduces the slice size.

One characteristic of APNs that makes them complex to slice is the use of multiset of algebraic terms over the arcs. In principle, algebraic terms may contain the variables. Even though, we want to reach a syntactically reduced net, its reduction by slicing needs to determine the *reading and non-reading* semantic nature of transitions. For this we need to analyze the possible ground substitutions of these algebraic terms.

We propose to unfold the APN first and then perform the slicing on the unfolded APN. In general, unfolding generates all possible firing sequences from the initial marking of the APN, though maintaining a partial order of events based on the causal relation induced by the net, concurrency is preserved. AlPiNA (a symbolic model checker for Algebraic Petri nets) allows the user to define partial algebraic unfolding and presumed bounds for infinite domains [1], using some aggressive strategies for reducing the size of large data domains. Therefore, we follow the partial algebraic unfolding approach in this work [1]. The notion of unfolding for APNs is out of the scope, for further details and description about algebraic unfolding used in our approach, we refer the interested reader to follow [9]. To describe formally our APN slicing technique, we provide below the necessary definitions:

Definition 2. *Let N be an unfolded APN and $t \in T$ be a transition. We call t a reading-transition iff its firing does not change the marking of any place $p \in ({}^\bullet t \cup t^\bullet)$, i.e., iff $\forall p \in ({}^\bullet t \cup t^\bullet), \lambda(p,t) = \lambda(t,p)$. Conversely, we call t a non-reading transition iff $\lambda(p,t) \neq \lambda(t,p)$.*

The distinction between *reading and non-reading transitions* is based on the syntactic equivalence between the multiset of terms over the arcs. In the Fig.2, we give an overview of the proposed approach for slicing based verification of APNs using Process Flowchart. At first, the APN model is unfolded and then by taking the property into an account *criterion places* are extracted. Afterwards, slicing is performed for the *criterion places.* Subsequently, verification is performed on the sliced unfolded APNs. The user may use the counterexample provided by the model checker used to refine the APN model to correct the property or change the APN model.

3.1 Example: Unfolding an APN

Fig. 3 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 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

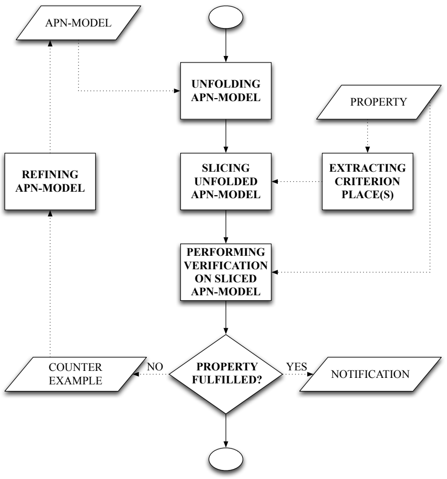

Fig. 2. Process Flowchart of *slicing* based verification of APN models

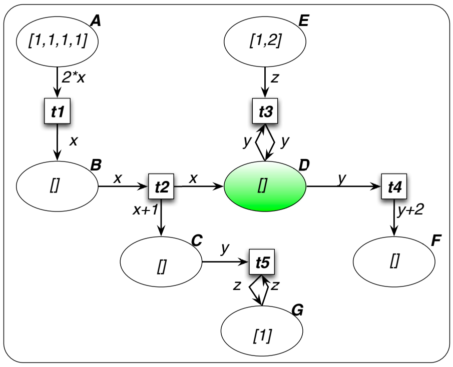

Fig. 3. An example APN model (*APNexample*)

ever assume any natural value above 3. For this reason, following [1], we can set a *presumed bound* of 3 for the *naturals* data type, greatly reducing the size of the data domain.

The resulting unfolded APN model is shown in Fig. 4. The transitions arcs are indexed with the incoming and outgoing values of tokens. The proposed process assumes that an unfolding takes place before 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 global complexity of the process. After having defined the slicing algorithm, we make an extremely simple example of how the slicing algorithm works, starting from an APN, unfolding it and slicing it.

3.2 The Slicing Algorithm

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 1: APN slicing algorithm for unfolded APN model

```
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 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.3, let us now take an example property and apply our proposed algorithm on it. Informally, we can define the property:

"The values of tokens inside place D are always smaller than 5".

Formally, we can specify the property in *LTL* as $\mathbf{G}(\forall tokens \in D | tokens < 5)$. For this property, the slicing criterion $Q = \{D\}$, as D is the only place concerned by the property. Therefore, the application of *APNSlicing(UnfoldedAPN, D)* returns *SlicedUnfoldedAPN* (shown in Fig. 5), which is smaller than the original *UnfoldedAPN* shown in Fig. 4).

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 have 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.

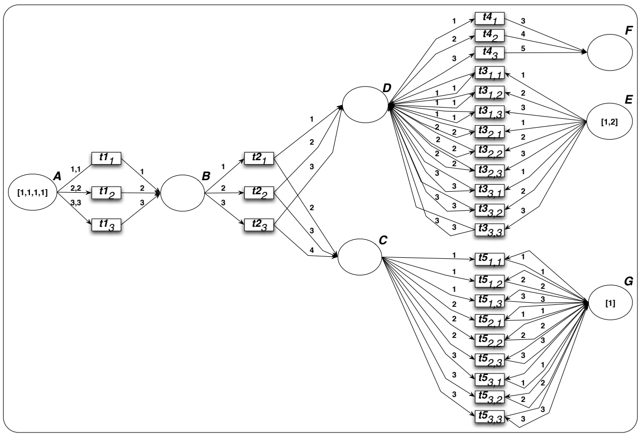

Fig. 4. The unfolded example APN model (*UnfoldedAPN*)

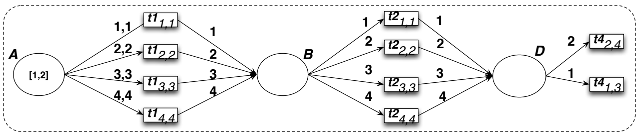

Fig. 5. Sliced and Unfolded example APN model (*SlicedUnfoldedAPN*)

4 Verification of APNs Evolutions

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. APNs formalism consists of the control aspect that is handled by a Petri net and a data aspect that is handled by one or many abstract algebraic data types(AADTs). In general, the evolutions to APNs can be divided into two parts,

– Evolutions to the structural aspect
– Evolutions to 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 can evolve with respect to the structural changes such as: add/remove places, transitions, arcs, tokens and terms over the arcs. By notation, different APNs will be noted with superscripts such as $APN' = \langle SPEC, P', T', F', asg', cond', \lambda', m'_0 \rangle$. As there is no guarantee that after every evolution of the APN model, it still satisfies the previously satisfied properties, model checking is repeated after every evolution, which is very expensive in terms of time and space.

To avoid the repeated model checking, we propose a slicing based solution to reason about the previously satisfied properties. At first, after the evolution of the APN model, we build the slices for the evolved and non-evolved APN models with respect to the property by the *APNSlicingEvo* algorithm. The algorithm is somewhat similar to the *APNSlicing* algorithm given in the Section 3. The main difference is that it does not exclude the *reading transitions* from the slice (evolutions to *reading transitions* can disturb the property satisfaction) and it takes APN model instead of *unfolded* APN model as an input (reducing the overhead of unfolding). The slicing algorithm starts with an APN model 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 APN model based on Q, using following algorithm:

Algorithm 2: APN slicing algorithm for APN model

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

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

4.1 Classification of Evolutions

As discussed above, we build the slices for the evolved and non-evolved APN models with the help of *APNSlicingEvo* algorithm. Once we build the slices, by comparing both APN models, we can divide the evolutions into two major classes as shown in the Fig.6. 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 verification cost and time.

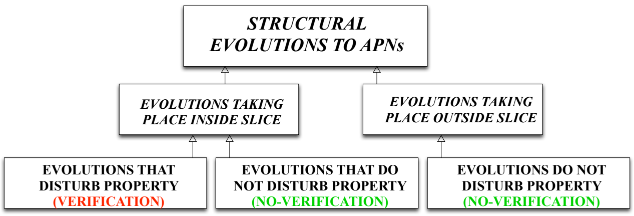
STRUCTURAL EVOLUTIONS TO APNs
EVOLUTIONS TAKING PLACE INSIDE SLICE
EVOLUTIONS TAKING PLACE OUTSIDE SLICE
EVOLUTIONS THAT DISTURB PROPERTY (VERIFICATION)
EVOLUTIONS THAT DO NOT DISTURB PROPERTY (NO-VERIFICATION)
EVOLUTIONS DO NOT DISTURB PROPERTY (NO-VERIFICATION)

Fig. 6. Classification of structural evolutions to APNs

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 the model is analyzed for. 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 1. *Let* $N = \langle SPEC, P, T, F, asg, cond, \lambda, m_0 \rangle$ *be a sliced APN and* $N' = \langle SPEC, P', T', F', asg', cond', \lambda', m'_0 \rangle$ *be an evolved sliced APN′ w.r.t the property* φ. $N \models \varphi \Rightarrow N' \models \varphi$ *if and only if*
1) $P = P' \wedge \forall p \in (P \cap P') \mid m_0(p) = m'_0(p) \wedge asg(p) = asg'(p)$,
2) $T = T' \wedge \forall t \in (T \cap T') \mid cond(t) = cond'(t)$,
3) $\forall p \in (P \cap P'); \forall t \in T \mid ((p,t) \in F) \Rightarrow (t \in T' \wedge (p,t) \in F' \wedge \lambda(p,t) = \lambda'(p,t))$,
4) $\forall p \in (P \cap P'); \forall t \in T \mid ((t,p) \in F) \Rightarrow (t \in T' \wedge (t,p) \in F' \wedge \lambda(t,p) = \lambda'(t,p))$.

For all the proofs of theorems given in this work, we refer the interested reader to [7]. Let us recall the APN model and the example property given in the section 3. The example property is following $G(\forall token \in D | tokens < 5)$. Fig.7, shows some possible examples of the evolutions to APN 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 doted lines (remark that we follow the same convention for all examples). In the first evolution example, algebraic terms are changed and shown with the red color. In the second example, values of tokens are 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.

Evolutions Taking Place Inside the Slice: For all the evolutions that are taking place inside the slice, we divide them into two classes i.e., evolutions that require verification or not. Identifying such class of evolutions is extremely hard due to non-determinism of the possible evolutions. Specifically, in APNs small structural changes can impact the behavior of the model. It is also hard

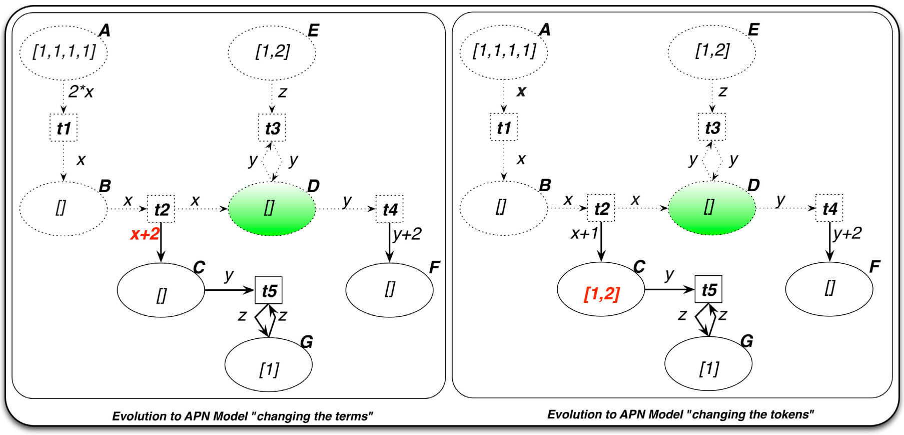

Fig. 7. Evolutions to APN model taking place outside the slice

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 the specific evolutions. For an example, for all the safety properties specified by the temporal formula $\mathbf{G}(\varphi)$ *or* $\exists\mathbf{G}(\varphi)$, if φ an atomic formula, using the ordering operators $\leq$ *or* $<$ between the *places* and their *cardinality* or tokens inside *places* and their *values*, then all the evolutions that decrease the tokens do not require verification because they do not impact the behavior required for the property satisfaction.

Theorem 2. *Let* $N = \langle SPEC, P, T, F, asg, cond, \lambda, m_0\rangle$ *be a sliced APN and* $N' = \langle SPEC, P', T', F', asg', cond', \lambda', m'_0\rangle$ *be an evolved sliced APN′ (in which tokens are decreased) w.r.t the property* φ*. For all the safety properties specified by temporal formulas i.e.,* $\boldsymbol{G}(\varphi) or \exists\boldsymbol{G}(\varphi)$*, and* φ *a forumla using* $\leq$ *or* $<$ *ordering operator between the places and their cardinality or tokens inside places and their values.* $N \models \varphi \Rightarrow N' \models \varphi$ *if and only if*

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

2) $T = T' \wedge \forall t \in (T \cap T') \mid cond(t) = cond'(t)$,

3) $\forall p \in (P \cap P'); \forall t \in T \mid ((p,t) \in F) \Rightarrow (t \in T' \wedge (p,t) \in F' \wedge \lambda(p,t) \geq \lambda'(p,t))$,

4) $\forall p \in (P \cap P'); \forall t \in T \mid ((t,p) \in F) \Rightarrow (t \in T' \wedge (t,p) \in F' \wedge \lambda(t,p) \geq \lambda'(t,p))$.

Let us recall the APN model and the example property given in the Section 3. The example property is following $\mathbf{G}(\forall token \in D | tokens < 5)$, we can avoid the verification for several evolutions even if they are taking place inside the slice. Some possible examples of the evolutions are shown in Fig.8. In the first example

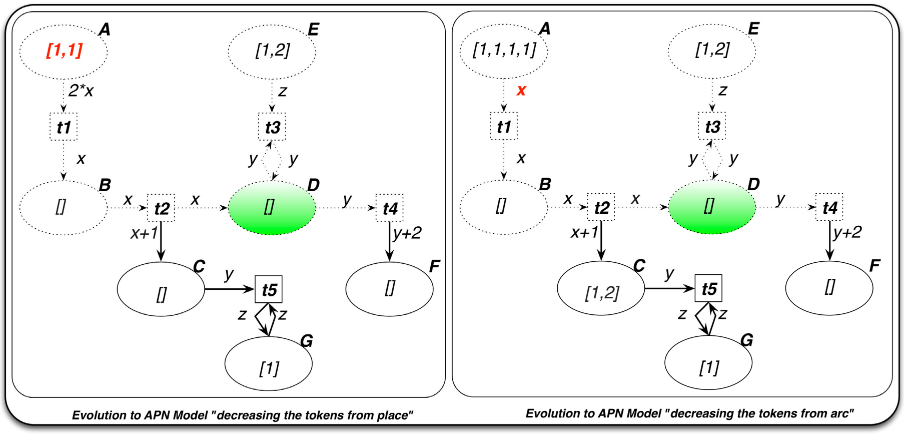

Fig. 8. Evolutions to APN model taking place inside the slice

tokens are decreased from a place and in the second example tokens are decreased from an arc, but the property is still satisfied.

For all the liveness properties specified by the temporal formula $\exists\mathbf{F}(\varphi)$, 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, it is not required to verify them as they do not impact the behavior required for the property satisfaction.

Theorem 3. *Let* $N = \langle SPEC, P, T, F, asg, cond, \lambda, m_0\rangle$ *be a sliced APN and* $N' = \langle SPEC, P', T', F', asg', cond', \lambda', m'_0\rangle$ *be an evolved sliced APN' (in which tokens are increased) w.r.t the property* φ*. For all the liveness properties specified by temporal formula* $\exists\boldsymbol{F}(\varphi)$*, and* φ *is using the ordering operators* $\geq$ *or* $>$ *between the* places *and their* cardinality *or tokens inside* places *and their* values*.* $N \models \varphi \Rightarrow N' \models \varphi$ *if and only if*

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

2) $P = P' \wedge \forall t \in (T \cap T') \mid cond(t) = cond'(t)$,

3) $\forall p \in (P \cap P'); \forall t \in T \mid ((p,t) \in F) \Rightarrow (t \in T' \wedge (p,t) \in F' \wedge \lambda(p,t) \leq \lambda'(p,t))$,

4) $\forall p \in (P \cap P'); \forall t \in T \mid ((t,p) \in F) \Rightarrow (t \in T' \wedge (t,p) \in F' \wedge \lambda(t,p) \leq \lambda'(t,p))$.

Let us consider again the APN model given in Section 3, if we are interested to verify the example property such as: $\exists\mathbf{F}(|D| > 3)$, verification can be avoided completely for several evolutions even if they are taking place inside the slice. Some possible examples of the evolutions are shown in Fig.9. In the first and second examples, tokens are increased but the property is still satisfied.

We identified above that for several specific evolutions and properties verification could be completely avoided, and for the rest of evolutions we can perform verification only on the part that concerns the property by following Section 3.

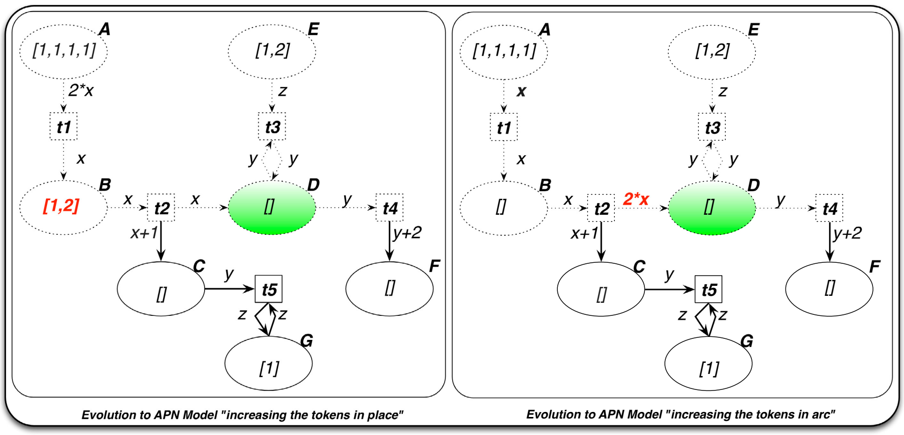

Fig. 9. Evolutions to APN model taking place inside the slice

Even in this case we significantly improve the verification of evolution. The theory developed has been applied to the car crash management system case study. We refer the interested reader to [7] for the case study.

5 Related Work

We first highlight the differences and similarities related to state space reductions. Secondly, we discuss the related work regarding the optimization of verification with respect to evolving system nets.

5.1 Petri Net Reductions

Model checking is a convenient approach for the analysis of concurrent and distributed systems. Its main disadvantage is the so-called state space explosion problem where the size of the state space can grow exponentially in the size of the system. The problem of the optimizing model checking process has been extensively studied in the literature. Several dedicated approaches are proposed to alleviate the state space exploration by using efficient data structure such as binary decision diagram [2]. One of the possible approaches to ameliorate the model checking is modular analysis [4]. The internal activity of the modules is explored independently rather than in an interleaved fashion. The idea is to generate a state space for each module and the information compulsory to capture the interaction between modules, and in this way the construction of full state space is avoided. In the context of Petri nets there exist different approaches for modularizing . In [5] authors presented two different ways for modularizing simple petri, one using place sharing, and the other using transition sharing. They showed how place invariants and state spaces could be constructed in a modular way. In [10] authors investigated the possibility of using modular state spaces for simple Petri nets when modules communicate through place fusion. This work provides some guidelines, which could be partially automated but they are quite general and should not be considered as always giving the best results.

Slicing is a technique used to reduce a model syntactically. The reduced model contains only those parts that may affect the property the model is analyzed for. Slicing Petri nets is gaining much attention in the recent years [3,9,14,16–18]. Mark Weiser introduced the slicing term in [20], and presented slicing as a formalization of an abstraction technique that experienced programmers (unconsciously) use during debugging to minimize the program. The first algorithm about Petri net slicing was presented by chang et al [3]. They proposed an algorithm on Petri nets testing that slices out all sets of paths, called concurrency sets, such that all paths within the same set should be executed concurrently. Lee et al. proposed the Petri nets slice approach in order to partition huge place transition net models into manageable modules, so that the partitioned model can be analyzed by compositional reachability analysis technique [12].

Astrid Rakow developed two notions of Petri net slicing, CTL^*_{-X} slicing and *Safety slicing* in [18]. The key idea behind the construction is to distinguish between reading and non-reading transitions. A reading transition $t \in T$ can not change the token count of place $p \in P$ while other transitions are non-reading transitions. For CTL^*_{-X} slicing, a subnet is built iteratively by taking all non-reading transitions of a place P together with their input places, starting with given criterion place. For the *Safety slicing* a subnet is built by taking only transitions that increase token count on places in P and their input places. CTL^*_{-X} slicing algorithm is fairly conservative. By assuming a very weak fairness assumption on Petri net it approximates the temporal behavior quite accurately by preserving all CTL^*_{-X} properties and for safety slicing focus is on the preservation of stutter-invariant linear safety properties only.

In [9], we introduced the Algebraic Petri net slicing for the first time. We adapt the notion of *reading and non-reading transitions* in the context of Algebraic Petri nets defined for the low-level Petri nets. The main difference between the existing slicing constructions such as, CTL^*_{-X}, *Safety slicing* and our is that in CTL^*_{-X}, *Safety slicing* only transitions are included that change the token count whereas in *APNSlicing*, we include transitions that change the token values together with the transitions that change the token count.

5.2 Property Preserving Petri Nets Evolutions

Most of the work regarding optimizing the 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, building a full categorical framework for APNs, the rule-based refinements [6,13,15]. Padberg consider the notion of rule-based modification of Algebraic high level nets preserving the safety properties. The theory of 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. In contrast to transition preserving morphisms in [15], it preserves the safety properties by extending the rule-based modification of APNs. These morphisms, called the place preserving morphisms, allow transferring specific temporal logic formulas expressing net properties from the source to the target net. Lucio presented a preliminary study on the invariant preservation of behavioral models expressed in Algebraic Petri nets, in the context of an iterative modeling process [13]. They proposed to extend the property preserving morphisms in a way that it becomes possible to strengthen the guards without loosing previous behaviours.

In contrast to the property preservation, the scope of our work is broader. At first, we try to find out which evolutions require verification independent of temporal representations of the properties. Secondly, we focus on the specific properties and evolutions to optimize the verification in general. To give more flexibility to the user, we do not restrict the type of evolutions and properties. It is important to note that our proposed technique can further refine the previous proposals about the property preservation. The proposal is to preserve the morphisms restricted to the sliced part of the net.

6 Conclusion and Future Work

In this work, we developed an approach to optimize the verification of structurally evolving APNs. The proposed work is based on the slicing and pursues two goals; the first is to perform verification only on the parts that may affect the property the APN model is analyzed for. The second is to classify the evolutions of APNs 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 APNs model checking and the verification of structural evolutions of APNs.

The 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 structurally evolving APNs based on the slicing and the classification of structural evolutions proposed in this work.

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

References

1. Buchs, D., Hostettler, S., Marechal, A., Risoldi, M.: AlPiNA: A symbolic model checker. In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 287–296. Springer, Heidelberg (2010)
2. Burch, J.R., Clarke, E., McMillan, K.L., Dill, D., Hwang, L.J.: Symbolic model checking: 1020 states and beyond. In: Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, LICS 1990, pp. 428–439 (1990)

3. Chang, J., Richardson, D.J.: Static and dynamic specification slicing. In: Proceedings of the Fourth Irvine Software Symposium (1994)
4. Christensen, S., Petrucci, L.: Towards a modular analysis of coloured petri nets. In: Jensen, K. (ed.) ICATPN 1992. LNCS, vol. 616, pp. 113–133. Springer, Heidelberg (1992)
5. Christensen, S., Petrucci, L.: Modular analysis of petri nets. The Computer Journal 43, 2000 (2000)
6. Er, S.P.: Invariant property preserving extensions of elementary petri nets. Technical report, Technische Universitat Berlin (1997)
7. Khan, Y.I.: Optimizing verification of structurally evolving algebraic petri nets. Technical Report TR-LASSY-13-03, University of Luxembourg (2012)
8. Khan, Y.I., Risoldi, M.: Language enrichment for resilient MDE. In: Avgeriou, P. (ed.) SERENE 2012. LNCS, vol. 7527, pp. 76–90. Springer, Heidelberg (2012)
9. Khan, Y.I., Risoldi, M.: Optimizing algebraic petri net model checking by slicing. In: International Workshop on Modeling and Business Environments (ModBE 2013, associated with Petri Nets 2013) (2013)
10. Lakos, C., Petrucci, L.: Modular state spaces and place fusion. In: International Workshop on Petri Nets and Software Engineering (PNSE 2007, associated with Petri Nets 2007), pp. 175–190 (2007)
11. Larman, C., Basili, V.: Iterative and incremental developments. A brief history. Computer 36(6), 47–56 (2003)
12. Lee, W.J., Kim, H.N., Cha, S.D., Kwon, Y.R.: A slicing-based approach to enhance petri net reachability analysis. Journal of Research Practices and Information Technology 32, 131–143 (2000)
13. Lucio, L., Syriani, E., Amrani, M., Zhang, Q., Vangheluwe, H.: Invariant preservation in iterative modeling. In: Proceedings of the ME 2012 Workshop (2012)
14. Llorens, M., Oliver, J., Silva, J., Tamarit, S., Vidal, G.: Dynamic slicing techniques for petri nets. Electron. Notes Theor. Comput. Sci. 223, 153–165 (2008)
15. Padberg, J., Gajewsky, M., Ermel, C.: Rule-based refinement of high-level nets preserving safety properties. In: Astesiano, E. (ed.) ETAPS 1998 and FASE 1998. LNCS, vol. 1382, pp. 221–238. Springer, Heidelberg (1998)
16. Rakow, A.: Slicing petri nets with an application to workflow verification. In: Geffert, V., Karhumäki, J., Bertoni, A., Preneel, B., Návrat, P., Bieliková, M. (eds.) SOFSEM 2008. LNCS, vol. 4910, pp. 436–447. Springer, Heidelberg (2008)
17. Rakow, A.: Slicing and Reduction Techniques for Model Checking Petri Nets. PhD thesis, University of Oldenburg (2011)
18. Rakow, A.: Safety slicing petri nets. In: Haddad, S., Pomello, L. (eds.) PETRI NETS 2012. LNCS, vol. 7347, pp. 268–287. Springer, Heidelberg (2012)
19. Reisig, W.: Petri nets and algebraic specifications. Theor. Comput. Sci. 80(1), 1–34 (1991)
20. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering, ICSE 1981, pp. 439–449. IEEE Press, Piscataway (1981)