A Method to Analyze High Level Petri Nets using SPIN Model Checker

Dewan Mohammad Moksedul Alam
School of Computing and Information Sciences
Florida International University
Miami, Florida 33199, USA
dalam004@fiu.edu

Xudong He
School of Computing and Information Sciences
Florida International University
Miami, Florida 33199, USA
hex@cis.fiu.edu

Abstract— **High level Petri nets (HLPNs) are a formal method for studying concurrent and distributed systems and have been widely used in many application domains. However, their strong expressive power hinds their analyzability. In this paper, we present a new transformational analysis method for analyzing a special class of HLPNs – predicate transition (PrT) nets. This method extends and improves our prior results by covering more PrT net features including full first order logic formulas and exploring additional alternative translation schemes. This new analysis method is supported by a tool chain – front end PIPE+ for creating and simulating PrT nets and back end SPIN for model checking safety and liveness properties. We have applied this method to two benchmark systems used in annual Petri net model checking contest 2015. We discuss several properties and show the detailed model checking results of two properties in one system.**

Keywords— formal methods; high level Petri nets; temporal logic; model checking; SPIN

I. INTRODUCTION

High level Petri nets (HLPNs) are a powerful formal method for modeling concurrent and distributed systems. HLPNs provide a graphical representation of systems to make them easier to understand. HLPNs offer strong expressive power through rich data abstraction, and algebraic expressions and logic formulas for defining system functionality. Furthermore, the dynamic semantics of HLPNs supports model level simulation. As a result, HLPNs are being used widely in system modeling in many application domains.

However, due to the expressive power of HLPNs, they are difficult to analyze. This is especially critical when we need to study the safety and liveness properties of highly dependable systems modeled in HLPNs. In recent years, a variety of new analysis techniques based on model checking have been proposed to analyze high level Petri nets in addition to traditional techniques such as simulation. These new analysis techniques fall into two general categories: (1) developing tailored model checking algorithms for particular types of HLPNs, or (2) leveraging existing general model checkers through model translation where a HLPN is transformed into an equivalent form suitable for the target model checker.

Although tailored model checkers for HLPNs may take advantages of the unique features of the underlying HLPNs and perform specific optimizations to improve performance, they often lack the full-fledged features provided by the well-known general model checkers and are not easily adaptable. Furthermore, there are no convincing experiments yet to show tailored model checkers have performance advantages over using general model checkers. Several tools supporting direct model checking of HLPNs participated in the annual Petri net model checking contest, but had very limited success.

An alternative way of analyzing HLPNs leverages existing well-known general purpose model checkers. A translation method is needed to generate a state transition system from a given HLPN. Model translations have been used and explored for many years [1]. The key of a translation method is to ensure the behavioral equivalence of the source model and target model. It is quite straightforward to obtain a state transition system from a given HLPN with the loss of true concurrency. Fortunately, safety and liveness properties are preserved from all known Petri net semantics – causal (partial order), maximal concurrency, interleaving set, and interleaving. In our prior work ([2], [3], [4]), we developed methods to translate a particular type of HLPNs – predicate transition nets (PrT nets) into the input models of several well-known model checkers including SMV [5] and SPIN [6] with various limitations. In this paper, we present a new and more general method to translate PrT nets to SPIN. This method includes the following new contributions: (1) eliminating many limitations such as no set variables on arc label in our prior work, (2) supporting advanced features including quantifiers in full first order logic formulas, (3) combining transition enabling testing and firing to improve performance, and (4) offering an additional process based translation scheme. These new features have been implemented in our enhanced PIPE+ environment previously reported in [7]. We demonstrate the application of our method through two benchmark systems used in the annual model checking contest of Petri net tools 2015 [8].

II. PREDICATE TRANSITION NETS

In the following sections, we give a brief introduction to PrT nets. More detailed definitions of PrT nets can be found in [9].

A. The Syntax and Static Semantics of PrT Nets

A PrT net is a tuple *(N, Spec, ins)* where $N = (P, T, F)$ is a net structure, in which (1) P and T are disjoint nonempty finite sets (the sets of places and transitions of N respectively), (2) $F \subseteq P \times T \cup T \times P$ is a flow relation (the arcs of N). *Spec = (S,*

DOI reference number: 10.18293/SEKE2017-162

Op, Eq) is the underlying algebraic specification, and consists of a signature $\Sigma = (S, Op)$ and a set *Eq* of Σ-equations. Signature Σ includes a set of sorts *S* and a family $Op = (Op_{s1,s2\ldots sn;s})$ of sorted operations for $s1, \ldots, sn, s \in S$. For $s \in S$, we use Con_s to denote *Op;s* (the 0-ary operation of sort *s*), i.e. the set of constant symbols of sort *s*. The Σ-equations in *Eq* define the meanings and properties of operations in *Op*. *Spec* is a meta-language to define the tokens, labels, and constraints of a PrT net. Tokens of a PrT net are ground terms of signature Σ, written $MCon_S$. The set of labels is denoted by $Label_S(X)$ (*X* is the set of sorted variables disjoint with *Op*). Each label can be a simple variable or a set expression of the form $\{x_1, x_2, \ldots, x_n\}$. Constraints of a PrT net are a subset of first order logic formulas (where the domains of quantifiers are finite and any free variable in a constraint appears in the label of some connecting arc of the transition). The subset of first order logical formulas contains the Σ-terms of sort *Bool* over *X*, denoted as $Term_{(Op;Bool)}(X)$.

$Ins = (\varphi, L, R, M_0)$ is a net inscription that associates a net element in *N* with its denotation in *Spec*. $\varphi : P \to \wp(S)$ is the data definition of *N* and associates each place p in *P* with a subset of sorts in *S*. $L : F \to Label_S(X)$ is a sort-respecting labeling of PrT net. We use the following abbreviation in the following definitions: $\bar{L}(x, y) = L(x, y)$ if $(x, y) \in F$ or $\bar{L}(x, y) = \emptyset$ if $(x, y) \notin F$. $R : T \to Term_{(Op;Bool)}(X)$ is a well-defined constraining mapping, which associates each transition *t* in *T* with a first order logic formula defined in the underlying algebraic specification. Furthermore, the constraint of a transition defines the meaning of the transition. We use *var(t)* to denote variables appearing in *R(t)*. $M_0 : P \to MCons_S$ is a sort-respecting initial marking. The initial marking assigns a multiset of tokens to each place *p* in *P*.

A Σ-algebra of *Spec* provides interpretations for the sorts and operations in *Spec*, which includes familiar sorts such as integer, Boolean, string, tuple, and set as well as their relevant operations and equations. In our tool environment, the Σ-algebra is instantiated with a subset of Java data types and their associated operations and laws.

B. The Dynamic Semantics of PrT Nets

The dynamic semantics of PrT nets are defined on the concept of markings (states) that are mappings from places to tokens. A transition *t* in *T* is *enabled* in a marking *M* if there are tokens in its input places that satisfy its precondition defined in constraint *R*(*t*). Formally, the enabling condition can be defined as: $\forall p \in P. (\bar{L}(p, t) : \alpha \subseteq M(p) \wedge R(t) : \alpha)$, where α is an instantiation of arc variables with tokens in p. An enabled transition *t* can *fire* by removing tokens from its input places and adding tokens to its output places according to the post condition defined in constraint *R*(*t*), which results in a new marking *M'*. Formally, transition firing can be defined as: $M'(p) = M(p) \cup \bar{L}(t, p) : \alpha - \bar{L}(p, t) : \alpha$ for all $p \in P$, where α is an instantiation satisfying $R(t) : \alpha$. Two enabled transitions may fire at the same time as long as they are not in conflict, i.e. the firing of one them disables the other. The dynamic semantics (behavior) of a PrT net is the set of all possible transition firing sequences starting from the initial marking. The essential dynamic semantics regarding the translation method are discussed later.

III. THE MODEL CHECKER SPIN

Promela is the underlying modeling language for SPIN to describe a system model. An operational model in Promela contains one or more *process*es, zero or more *variables*, zero or more *message channels* and a *semantics engine* [6].

Processes are the central construct in a Promela model to define system behaviors, and are defined using *proctype* declaration.

Message channels are used to model the transfer of data from one state to another, which can store a finite number of messages as declared. Apart from storing data, channels provide a wide range of features to model message passing in a clean and efficient way. Basically, they are FIFO queue but can also be used for random accesses. When a new message is sent to a channel, it is added to the end. When an attempt is made to retrieve a message, it always returns a message in front. It is also possible to query a channel for a specific message. By default, channel removes messages as they are retrieved. However, it is also possible to get a message without removing it. Some basic channel usages are given below.

Table 1- Basic usage of message channel

```
1.  chan qname = [8] of {int, short}
2.  chan qname = [8] of {s_type}
3.  qname!10, 20
4.  qname?x, y
5.  qname?x, eval(y)
6.  qname?[x, eval(y)]
7.  qname??[x, eval(y)]
8.  len(qname)
```

In the above example, lines 1 and 2 show the declaration of channels with basic datatypes and structured datatypes respectively. Line 3 sends message to the channel. Lines 4-7 show different ways to retrieve messages. Lines 4, 5 and 6 retrieve first message. Lines 4 and 5 also remove the message after retrieving. Lines 5 and 6 check whether the second element of the first message matches the value currently hold in y. Line 7 searches for a match anywhere in the channel. Lines 6 and 7 retrieve a message without removing it (poll operation). Line 8 returns the number of messages in the channel.

Promela provides non-determinism by default. The case selection and looping shown in Table 2 both have similar constructs. Each case is marked with a guarded statement (started with ::). If there are more than one case, then the sequence of statements will be selected for which the guarded statement is executable. If there are more than one such statement, then one of those will be chosen non-deterministically. If no such statement present, then the block is exited. A loop tries to find one executable guarded statement each time it completes execution. Promela also has a *for* loop construct, which is only to be used to iterate through channels.

Table 2 - Examples of control constructs

Case selection	Looping
if :: case 1 :: case 2 :: case 3 fi	do :: case 1 :: case 2 :: case 3 od

The next important thing is inline functions. Although Promela does not have general function concept, but supports inline functions as macros. We take advantage of in-line functions to structure our translated code.

IV. TRANSLATION OF PRT NETS TO PROMELA

A complete translation needs to translate all the components of a PrT net to equivalent constructs in the target language. Our new translation method eliminates many limitations in our prior work ([3], [4]) and supports many advanced features such as complex data structures and first order logic formulas. We have also combined transition enabling testing and firing to improve model checking performance. Furthermore, we have implemented an additional new process based translation scheme, which has a major impact in checking some safety and liveness properties that are otherwise not checkable using non-deterministic selection of in-line function based transition translation. We discuss the main translation rules as well as identified problems in the following sections.

A. Translation of Places

In PrT nets, *places* basically store information called tokens and thus represent the states of a net. Furthermore, PrT nets are a data flow computation model. State changes occur through token movements. The place concept perfectly matches the *channel* concept in Promela. The built-in functions of *channel* make it easy to query specific tokens for checking transition enabling condition and to add/remove tokens for transition firing. Each place in a PrT net is thus translated into a channel with the same type and the tokens of in the place are translated into messages stored in that channel in the initialization. Thus, for all $p \in P$, we have the following two lines as declarations.

```
#define bound_p const
chan    place_p = [bound_p] of {type_p}
```

Here, *type_p* is a data type in Promela resulted from the translation of the data type of place *p*, which will be discussed in the translation of data types.

B. Translation of Transitions

Transitions are the core components of dynamic semantics of a net and play the most important role in the execution as discussed in a prior section. The execution of a transition has two parts: testing its enabling condition and then firing it if enabled.

The translation of transitions constitutes the operational model in Promela. Each transition $t \in T$ is translated using the algorithm shown in Table 3. The techniques adopted for evaluating precondition and post-condition are discussed in the translation of transition constraints.

C. Translation of Arcs

Arcs are not translated directly but arc labels are important in determining the input and output variables of transitions during the translation of transition constraints (i.e. precondition and post-condition).

Table 3 – A general algorithm to translate a transition

```
inline check_is_enabled_and_fire_t() {
  for all combinations of values the input
  variables to t can take, do the following
    if preconditions are satisfied then
        compute postconditions
        compute new marking
        return;
  done
}
```

D. Translation of Data Definition

The data definition of places are multisets over the set of the sorts. These multisets are translated as structured types in Promela, where each sort in the data type definition is represented as a field in the structure.

PIPE+ supports only two basic sorts – string and number. These are used to define more complex data types for places through Cartesian product. To translate these complex data types, we first need to determine the corresponding representations of the sorts. Promela does not support strings, we choose *mtype* in Promela to represent strings. Each string token is treated as an enum constant in *mtype*. Promela does not support real numbers, but provides three types to support integers: `int`, `short`, and `byte`, in which the latter two help to reduce the state space. We use `short` as the default implementation and let a modeler select other choices during translation. For example, a place p with data type, $\varphi(p)$ = <*String, Number, String*> is translated into a Promela structured data type shown in Table 4.

Table 4 – An example of translation of data type definition

```
typedef struct type_p {
     mtype field1;
     short field2;
     mtype field3;
}
```

E. Translation of Transition Constraints

The constraint of a transition $t \in T$ is defined using a first order logic formula, which specifies the relationships among input and output variables of *t*. The constraint has two parts – precondition only involving input variables and the post-condition containing output variables.

1) Translation of operators

PIPE+ supports a variety or algebraic, relational, logical, and set operators. A PrT net uses a subset of these operators to define their behavior. Table 5 shows the supported operators in PIPE+.

Table 5 – Supported operators in PIPE+

Category	Operators	Data type
Algebraic	+, -, *, /, %	number
Relational	=, ≠, <, ≤, >, ≥	number,string
Logical	∧, ∨, ¬, →,↔	bool
Set	∈, ∉, ∪, \, |, :, ·	
Quantifiers	∀, ∃, ∄	

Most of these operators have their usual meanings. Some are overloaded, such as = operator. When = is used in a precondition, it is a comparison. When '=' is used in a post-condition, it acts as an assignment.

2) Translation of Preconditions

Each logic expression without quantifiers in a PrT net is translated into an equivalent form supported in Promela by using the corresponding operators. For example, expression $(x[1] \geq 3.5 \wedge y[1] \neq 10) \rightarrow z[1] = 2 * x[1] + 1$ is translated to the equivalent expression in Promela shown in Table 6.

Table 6 – An example of translation of expression

```
!(x.field1 >= 3.5 && y.field1 != 10)|| z.field1
== (2*x.field1+1)
```

However, the set operators and quantifiers in Table 5 do not have equivalent representations. The translation of quantifiers is given later. The translation of set operations is partially completed and is not further discussed in this paper since they are rarely used based on our experiences.

3) Translation of Post-conditions

Translation of post-conditions involve evaluation of the expressions and assignments of the values to the designated output variables. Alike preconditions, the evaluation is almost the same. Only difference is that, the evaluated values are assigned to the designated output variables. These output variables will be used as messages to corresponding output channels during new marking generation.

F. Translation of the Initial Marking

An initial marking defines the initial state of a PrT net. The structured tokens of each place are simply translated into send statements to the corresponding channel.

G. Translation of Quantifiers

PIPE+ supports two types of quantifiers – universal (∀) and existential (∃). A first order logic formula with quantifiers is called a quantified formula. Quantifiers are essentially used as a part of preconditions for seeking desirable input tokens. Thus we focus on how to translate these formulas in checking transition enabling conditions.

We identify different types of quantified formulas based on their structures, which are important in correctly handling the quantifiers. (1) Is a quantified formula within another formula (possibly another quantified formula), (2) Does a quantified formula contain global variables (arc annotations and those from their outer quantifiers), (3) Can a quantified formula be nicely separated into a conjunction of precondition and post-condition, (4) Does a quantified contain another quantified formula within it and so on.

For each quantifier, an inline function implementing an algorithm like what shown in Table 3 is generated. The difference is that, for an existential quantifier, as soon as the first combination of tokens satisfying the enabling condition, the loop exits. And for a universal quantifier, as soon as the first combination of tokens violates the enabling condition, the loop stops. The in-line functions for quantifiers have the following benefits: (1) easy management of the nested quantifiers. It is only a matter of calling the inline functions for the nested quantifier, (2) declaration, definition and scope management of the local and global variables can be resolved easily. The underlying contract of inline functions provided by Promela takes care of the above tasks.

Only care needed is to make sure unique names are used in these inline functions and the loop control variables have the correct names.

The in-line functions for quantifiers are called inside the innermost for loop before the *if* block in the corresponding precondition evaluation function shown in Table 3. This way, we can ensure the availability of the arc variables.

H. Putting Pieces Together

So far we have discussed the techniques of translating each component of PrT nets into Promela's constructs. To complete the translation, we need to provide an overall execution structure based on the dynamic semantics of PrT nets. There are two essential ways to select a transition firing in Promela – (1) as a passive in-line function to be non-deterministically selected in a Do loop within a centralized process. As long as there is an enabled transition, the loop continues; or (2) as an active *process*, which will be selected for execution by the SPIN runtime if the enabling condition is true. The first strategy was used in our prior work, where a transition's enabling condition checking and firing were executed separately, which not only had the slow performance but also could result in incorrect result due to conflict. We resolve this problem in our new implementation by combining transition enabling checking with firing, which has also improved performance. Furthermore, we have also implemented the second strategy. This new translation strategy results in much better model checking results in detecting violations of some safety properties and is important to check liveness properties when weak fairness assumptions are needed.

I. Translation Correctness

The correctness of the translation method covers the completeness and consistency. Completeness measures whether all features of PrT nets are translated into Promela. Our current translation covers all PrT features except a few set operations, which will be implemented in near future. Consistency refers to the equivalence between the dynamic behavior of a PrT net and the dynamic behavior of the translated Promela program. Of course, we ignore the concurrency transition firings in PrT nets that do not affect the satisfiability of safety and liveness properties that are essentially state based. Therefore, we only need to compare interleaved executions between a PrT net and the translated Promela program. Each interleaved execution starts from the initial marking and continues by firing one enabled transition at a time. Using an induction proof principle, we can only show (1) the initial marking is translated correctly, (2) each transition is translated correctly, i.e. the enabling condition and firing result are translated correctly, and (3) each enabled transition will be selected to fire. Step (1) is trivial true in our translation method. Step (2) is arguably true based on our careful design and extensive testing. Step (3) is true for both our overall model execution strategies discussed in the previous section. However, the formal proof of a general translation method is not easy, which is the reason that few compilers have been formally verified.

V. EXPERINEMT RESULTS

To evaluate our translation method, we have used two benchmark systems, Bridge and SafeBus, from the annual Petri net model checking contest 2015 [8]. These are the only high level Petri net models available in the contest, which were defined using colored Petri nets (a type of high level Petri nets). These systems are redefined using PrT nets in PIPE+. We only present and describe the Bridge model in this paper due to the page limit. The PrT models of the Bridge system and the SafeBus system with different parameters in xml format and the generated Promela models with properties in pml format as well as SPIN model checking results can be found at https://bitbucket.org/ptnet/pipe/.

The bridge system represents a system of a single lane bridge with an automatic controller for controlling the two-way through traffic. The model proposed contains three control parameters (V, P, N), where V is the number of vehicles on each side of the bridge trying to get to the other side, P is the maximum number of vehicles allowed on the bridge, and N is the maximum number of vehicles from the same side allowed to pass in a row. A PrT model of the system with parameters (4, 5, 2) is shown in Fig. 1. The net inscriptions are shown in Table 7 and 8.

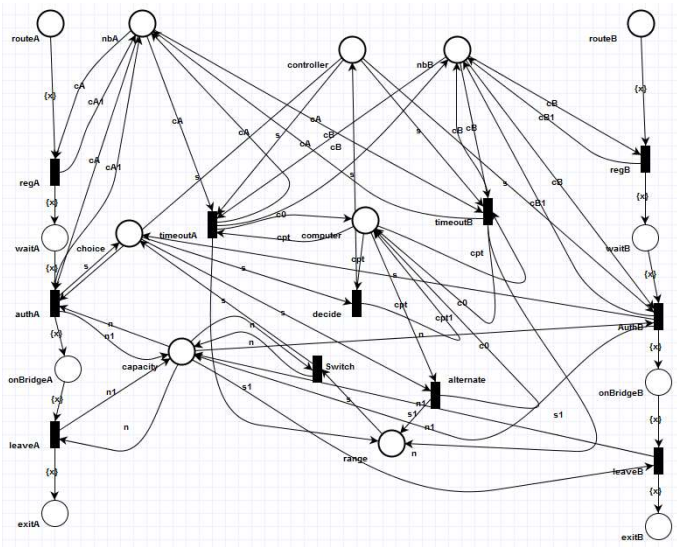

Fig. 1 – A PrT net model of the Bridge system

We have run simulations of PrT model in PIPE+ and simulations of the translated Promela model in SPIN. Both simulations ended with all vehicles crossed the bridge successfully. Although simulations in PIPE+ are much faster. The properties checked include: (1) the number of vehicles on the bridge never exceeds the maximal allowed (parameter P), (2) all the vehicles on the bridge must move in the same direction, (3) all the vehicles eventually cross the bridge, and (4) it cannot happen when some vehicles are at starting point and others have crossed the bridge. Properties (1) to (3) are the desirable ones – expected to hold, but property (4) is undesirable – expected to fail. These properties are specified using LTL expressions as (1)[]!((len(place_onBridgeA) > P || len(place_onBridgeB)) > P), where P is a constant based on the given model, (2) []!(len(place_onBridgeA)>0&&len(place_onBridgeB)>0), (3) for all x, [](routeA(x) -> <>exitA(x)) and a similar formula for the opposite direction B, and (4) []!(len(place_routeA)>0 && len(place_exitA)>0).

Table 7 – Data type definitions of the places

φ(routeA)=φ(waitA)=φ(exitA)=φ(onBridgeA)= ℘(number)
φ(routeB)=φ(waitB)=φ(exitB)=φ(onBridgeB)= ℘(number)
φ(nbA)=φ(nbB)=φ(timeoutA)=φ(timeoutB)= number
φ(controller)=φ(computer)=φ(capacity)= number
φ(choice)=φ(range)= number

Table 8 – Constraint definitions of the transitions

R(regA) = (cA1=cA+1)
R(authA)=((s=1∧n>0∧cA>0)∧(cA1=cA-1∧n1=n-1))
R(leaveA) = n1=n+1
R(timeoutA) = (s=1∧cA=0∧cB>0∧s1=2∧c0=0)
R(regB) = (cB1=cB+1)
R(authB)= ((s=2∧cB>0∧n>0)∧(cB1=cB-1∧n1=n-1))
R(leaveB)= n1=n+1
R(timeoutB) = (s=2∧cB=0∧cA>0∧s1=1∧c0=0)
R(decide) = (cpt<5∧cpt1=cpt+1)
R(alternate) = ((cpt=5∧c0=0∧s=1∧s1=2)∨ (cpt=5∧c0=0∧s=2∧s1=1))
R(switch) = (n=5)

In these experiments, properties (1) to (3) on all the Promela models up to (20,10,10) have been checked successfully without any violation, however due to the complexity of these models, SPIN quickly reached the allotted memory bound of 2048 Mbyte. Thus, only bounded state space is searched. The reported none violation can be false positive. Since all verification runs of the above properties in a particular model resulted in similar results except slight time differences since the state space explored is the same bounded by the allotted memory. A summary of SPIN model checking results of property (1) in the translated Promela models are shown in Tables 9 and 10. Table 9 shows the results where the transitions are translated as passive in-line functions to be non-deterministically selected to execute. Table 10 shows the results where transitions are translated as active processes. These experiments show that the first approach explores much smaller state space and is very fast than the second one. However, the second approach is effective in detecting violations in some safety properties, which found a counter example of property (4) in 47 milliseconds; while the first approach failed in finding a counter example in bounded search space as shown in Table 11. The second approach is also needed when a liveness property depends on weak fairness assumption since SPIN runtime environment will enforce fairness for processes.

Table 9 – Model checking results of property (1) using approach 1

Parameters (V, P, N)	State Transitions	Atomic steps	Search Depth	Time (seconds)
(4, 5, 2)	1479955	2437508	2823	2.99
(10,10,10)	1677016	2775407	5754	3.62
(20,10,10)	3670366	5990664	12730	10

Table 10 – Model checking results of property (1) using approach 2

Parameters (V, P, N)	State Transitions	Atomic steps	Search Depth	Time (seconds)
(4, 5, 2)	26172925	2.27E+08	4307	36.2
(10,10,10)	40727761	3.63E+08	7591	76.1
(20,10,10)	25392316	2.31E+08	14739	64

Table 11 – Checking results of property (4) with parameter (4, 5, 2)

Approach	States	Depth	Time	Result
1	9323528	2841	23.3	No-error
2	828	2351	0.047	Error

SPIN was not able to run large models due to state space explosion problem, which also happened the Petri model checking contest where none of the participating tools could verify the above high level Petri net models.

VI. Related work

During the past two decades, SPIN has been used as the analysis engine of many tool development efforts. SPIN has been used to model check programs. One of the most prominent work is Java PathFinder [11], which is a prototype translator from Java to Promela. Another work involves translation from C code to Promela [12]. SPIN has also been used as for model check specifications and designs. In [13], a formal approach was proposed to verify web service orchestration by translating the web service business process execution language (WS-BPEL) to Promela. Several studies [14, 15] attempted to formally verify the UML system models using SPIN by translating UML models to Promela.

Formal verification of Petri net models using SPIN have also been explored in the past several years. In [16], a simple technique was proposed to translate low level Petri nets to Promela. Several other similar techniques were proposed in the literature to translate low level Petri nets, but few works dealt with high level Petri nets. In our prior work [3, 4, 10], we proposed several techniques to translate PrT nets to Promela and implemented some of them with some restrictions in our tool environment PIPE+. Those concepts and their implementations suffer from some limitations – (1) those were not generic enough to support a wide range of system models, (2) did not support advanced features like quantifiers in full first order logic formulas, (3) suffered from some performance bottlenecks, (4) the translation scheme was rigid having no way to tweak the translation process, etc. are worth mentioning. Our new translation described in this paper eliminates these problems. The model translation feature in PIPE+ is fully automatic. Once a model is translated, we can explore powerful features of SPIN to verify the constraints and properties of the model using iSPIN graphical user interface.

VII. Conclusions

In this paper, we presented a method to translate PrT nets into a Promela programs. This new translation method supports many advanced features of PrT nets and provides a new execution scheme. The method is implemented in our tool environment PIPE+ and is completely automatic. Once a model is translated to Promela program, we can leverage SPIN's model checking capability to analyze system properties of PrT net models. Currently, we are doing more experiments to test SPIN's model checking capabilities, which will provide insights for model construction and specific PrT features to use. Furthermore, we want to develop a strategy to combine PrT's simulation capabilities and SPIN's model checking capabilities in analyzing different types of system properties when model checking may not be feasible. We will also utilize modeling knowledge to guide the translation, for example, mapping bounded integers in a PrT model to byte type in Promela, which may improve checking performance or make some property checking feasible. We will also fully implement the set operations in case some models require them. Our tool is open source and is available at https://bitbucket.org/ptnet/pipe/.

Acknowledgement

This work was partially supported by AFRL under FA8750-15-2-0106. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. We thank anonymous reviewers' comments for improving the presentation.

References

[1] S. Katz and O. Grumberg, "A Framework for Translating Models and Specifications," in Proceedings of the Third International Conference on Integrated Formal Methods, pp. 145–164, Springer-Verlag, 2002.

[2] X. He, H. Yu, T. Shi, J. Ding, and Y. Deng, "Formally analyzing software architectural specifications using SAM," J. Syst. Softw., vol. 71, no. 1-2, pp. 11–29, 2004.

[3] G. Argote-Garcia, P. J. Clarke, X. He, Y. Fu, and L. Shi, "A Formal Approach for Translating a SAM Architecture to PROMELA," in SEKE, pp. 440–447, 2008.

[4] Lily Chang and Xudong He: "A Methodology to Analyze Multi-Agent Systems Modeled in High Level Petri Nets", International Journal of Software Engineering and Knowledge Engineering – IJSEKE, vol.25, no.7, 2015, 1199-1235.

[5] "The SMV System", http://www.cs.cmu.edu/~modelcheck/smv.html.

[6] G. Holzmann, "The Spin Model Checker: Primer and Reference Manual," Addison-Wesley Professional, 2003.

[7] Su Liu and Xudong He: "PIPE+Verifier - A Tool for Analyzing High Level Petri Nets", Proc. of the 27th International Conference on Software Engineering and Knowledge Engineering (SEKE15), Pittsburgh, July 6 – 8, 2015.

[8] "Model checking contest @ Petri Nets", http://mcc.lip6.fr/models.php.s.

[9] X. He: "A Comprehensive Survey of Petri Net Modeling in Software Engineering", International Journal of Software Engineering and Knowledge Engineering - IJSEKE, vol. 23, no. 5, 2013, 589-626.

[10] Su Liu, Reng Zeng, Zhuo Sun, and Xudong He, "SAMAT - A Tool for Software Architecture Modeling and Analysis," Proc. of the 24th International Conference on Software Engineering and Knowledge Engineering, San Francisco, CA, pp. 352-358, July 2012.

[11] Havelund K: "Java PathFinder A Translator from Java to Promela". In: Dams D., Gerth R., Leue S., Massink M. (eds) Theoretical and Practical Aspects of SPIN Model Checking. SPIN 1999. Lecture Notes in Computer Science, vol 1680. Springer, Berlin, Heidelberg

[12] K. Jiang and B. Jonsson: "Using SPIN to Model Check Concurrent Algorithms, using a translation from C to Promela". In: Proc. 2nd Swedish Workshop on Multi-Core Computing, Uppsala, Sweden: Department of Information Technology, Uppsala University , 2009, 67-69.

[13] H. H. Kacem, W. Sellami, A. H. Kacem. "A Formal Approach for the Validation of Web Service Orchestrations". 2012 IEEE 21st International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises. June 2012. doi: https://doi.org/10.1109/WETICE.2012.53.

[14] L. Ji, J. Ma and Z. Shan: "Research on Model Checking Technology of UML". 2012 International Conference in Computer Science and Service System, p.p. 2337 - 2340, 2012.

[15] Y. Yamada and K. Wasaki: "Automatic generation of SPIN model checking code from UML activity diagram and its application to Web application design", The 7th International Conference on Digital Content, Multimedia Technology and its Applications, p.p. 139 - 144. 2011.

[16] G. C. Gannod and S. Gupta: "An automated tool for analyzing Petri nets using Spin", Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001), pages 404-407, 2001.