PIPE+Verifier - A Tool for Analyzing High Level Petri Nets

Su Liu and Xudong He
School of Computing and Information Sciences
Florida International University
Miami, Florida 33199, USA
{sliu002, hex}cis.fiu.edu

***Abstract*—High level Petri nets (HLPNs) have been widely used to model complex systems; however, their high expressive power costs their analyzability. Model checking techniques have been exploited in analyzing high level Petri nets, but have limited success due to either undecidability problem or state explosion problem. Bounded model checking (BMC) is a promising analysis method that explores state space within a predefined bound. BMC sacrifices the completeness of traditional model checking but becomes more practical and often effective to analyze large models. In our prior work, we have developed a method based on BMC and a supporting tool PIPE+Verifier to analyze high level Petri nets using a state of the art satisfiability modulo theories (SMT) solver Z3 as the backend engine. Our experiment results have been very encouraging. In this paper, we present the design, implementation, and use of PIPE+Verifier, as well as show additional improvements to make PIPE+Verifier more efficient.**

Keywords- Petri Net, Model Checking, Bounded Model Checking.

I. Introduction

High level Petri nets (HLPNs) [2] have been widely used to model the data, functionality, structure, and dynamic behaviors of complex systems. However the powerful expressiveness of HLPNs costs their analyzability. Simulations are the primarily analysis technique for HLPNs. Many HLPNs modeling tools such as CPN tools [10], [1], ALPiNA [9] and PIPE+ [12] support the simulation of different forms of HLPNs. While simulation is practical and cost effective, it cannot assure a safety property to be satisfied in all possible executions. Exhaustive analysis methods such as model checking [11] search all possible execution paths of a model but suffer from the state explosion problem, and are often limited to finite state systems. Since HLPNs can be used to model complex systems, where the state space can be not only huge but also infinite.

Bounded model checking (BMC) with satisfiability solving [6], [3] was proposed as an alternative approach to address the state explosion problem, which is particularly suited to analyze safety properties. BMC tries to find a counterexample violating a safety property by exploring only a finite state space defined by all execution paths up to a pre-defined bound k. A counterexample is found if the negated safety property is held in a reachable state; otherwise, the safety property holds

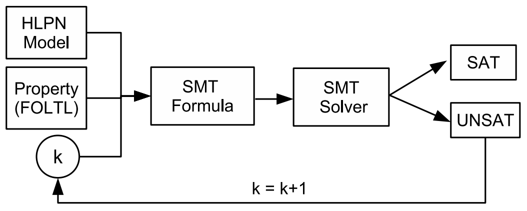

Figure 1: An Overview of PIPE+Verifier's Workflow

up to k. k can be iteractively increased to an acceptable value proportional to the size of a model's state space. Since the true upper bound cannot be determined in general, BMC is not a complete analysis method, yet is practical and effective in many real-world applications.

In SAT-based BMC [5], a model is converted into a propositional formula whose satisfiability is determined by a SAT solver. In recent years, satisfiability modulo theories (SMT) solvers [7] have made great progresses to efficiently check the satisfiability of a subset of first-order logic formulas with a variety of underlying theories including linear arithmetic, difference arithmetic, arrays and so on. These theories are rich enough to represent the data and algebraic expressions in most HLPN models.

In [13], we have developed a method based on BMC and a supporting tool PIPE+Verifier to analyze high level Petri nets using a state of the art satisfiability modulo theories (SMT) solver Z3 [8] as the backend engine. We have applied PIPE+Verifier to a variety of models from the existing literature and obtained very encouraging experimental results. An overview of PIPE+Verifier's workflow is shown in Figure 1.

PIPE+Verifier has the following features:

- being compatible with HLPN models and first-order linear time logic (FOLTL) representing properties built by modeling tool PIPE+ [12];
- encoding HLPN models and safety properties in FOLTL into an SMT formula;
- exporting the SMT formula into a file in C language (written in Z3's C API) recognizable by Z3;
- invoking Z3 to check the satisfiability of the SMT formula and returning an analysis report with the checking result, counterexample, consumed time and memory to check; and
- allowing incremental checking by increasing the k upper

DOI reference number: 10.18293/SEKE2015-060

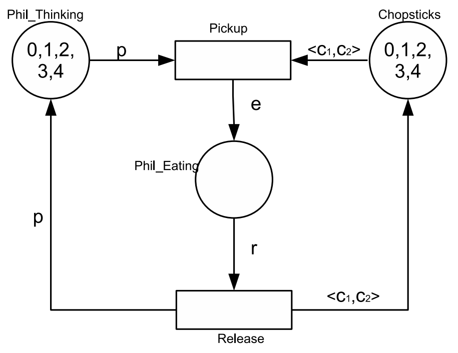

Figure 2: Five Dining Philosopher Problem in HLPN

bound value.

In this paper, we present the design, implementation, and use of PIPE+Verifier, as well as show additional improvements to make PIPE+Verifier more efficient with some experimental results.

II. Background

A. High Level Petri Nets

A HLPN [2] has a net structure consisting of a finite set of places (drawn as circles), a finite set of transitions (drawn as bars), and a finite set of directed arcs between places and transitions (drawn as arrows); and a net inscription supporting the definitions of place types, place markings, arc annotations, and transition conditions. A place type can be a power set to capture a set of tokens. All the tokens in a power set are of the same type built from primitive data types including integer type and string type. A place marking is a collection of tokens (data items) associated with the place. Arc annotations are inscribed with expressions that may comprise constants, variables, and function images. Transition conditions are logic expressions.

Figure 2 illustrates a dining philosopher problem modeled in a HLPN. The net consists of three places $P_{Phil\_Thinking}$, $P_{Chopsticks}$, P_{Phil_Eating} and two transitions T_{Pickup} and $T_{Release}$. All the places' token type is $\langle int \rangle$. $P_{Phil\_Thinking}$ and $P_{Chopsticks}$ both have five tokens initially $\{\langle 0\rangle, \langle 1\rangle, \langle 2\rangle, \langle 3\rangle, \langle 4\rangle\}$. T_{Pickup}'s transition condition is $p = c_1 \wedge (p+1)\%5 = c_2 \wedge e = p$. $T_{Release}$'s transition condition is $p = r \wedge c_1 = r \wedge c_2 = (r+1)\%5$.

B. Bounded Model Checking

Different from traditional model checking, BMC is incomplete and only performs an exhaustive search up to an upper bound. In many real world applications, a property can be effectively checked by examing only the limited prefixes of all executions, thus BMC becomes a practical and useful analysis technique, which partially alleviates the state explosion problem. Given a finite transition system M, a linear time temporal logic (LTL) formula f, and an integer k; BMC tries to determine whether there exists a computation path in M of length k or less (denoted as M_k) that satisfies f.

In BMC, a logic formula ϕ_k is constructed from a given M_k, including the initial state I and unrolled transition relation T, and some properties f. Since transition T in ϕ_k is unrolled k times, the length of ϕ_k is dependent on k. The logic formula ϕ_k is represented in equation 1:

$$\phi_k \doteq I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg f(s_i) \qquad (1)$$

where $I\,(s_0)$ is the characteristic function of the initial state, $T\,(s_i,\, s_{i+1})$ is the characteristic function of the transition relation, and $f(s_i)$ represents the property formula f associated with unrolled state s_i $(0 \leq i \leq k)$. Currently our tool supports the analysis of safety properties, thus f represents some safety property. If ϕ_k is satisfiable, there is a firing sequence or a state transition path from the initial state $I(s_0)$ to a state s_i that satisfies the negation of f_i, thus violates f; otherwise, property f holds in all execution sequences up to k transition steps in M .

Satisfiability modulo theories (SMT) [7] solvers are efficient modern theorem provers that support a combination of underlying theories such as bit-vectors, rational and integer linear arithmetic, arrays, and uninterpreted functions. SMT solvers are the extensions of satisfiability (SAT) solvers and directly applicable to the decision problems expressed in first order logic formulas with respect to the multiple background theories. For example, an SMT solver can decide whether a formula in the theory of linear arithmetic is satisfiable:

$$(x + y \leq 0) \wedge (\neg b \vee a \wedge (y = 0)) \wedge (x \leq 0)$$

where x, y are integer variables and a, b are Boolean variables. If the formula is satisfiable, the SMT solver returns a variable assignment satisfying the formula.

Both SAT solvers and SMT solvers have been successfully used in BMC. Z3 [8], developed in Microsoft Research Institution, is an efficient and widely used SMT solver that supports many background theories, such as rational and integer arithmetic, bit-vectors, array theory, and set theory. Z3 ranks highly in annual SMT competitions [4]. Therefore, Z3 is chosen as PIPE+Verifier's backend engine.

III. Translating HLPN Models to SMT Formulas

A. General Translation Rules HLPNs to a SMT Formulas

In BMC [6], a model and a property are encoded into a formula ϕ_k, which is solved by a SAT or SMT solver. Encoding a HLPN model and a property formula into ϕ_k involves the following steps.

1) Representing HLPN Markings as Symbolic States: In a HLPN model, a marking M_i is defined by a distribution of tokens in all places. Thus we need to define a symbolic state in SMT covering all places and their types in the HLPN. A mapping from HLPN model's elements to SMT sorts is shown in Table I.

A marking is defined by a SMT tuple with each tuple element denoting a place in the HLPN. Since a place can contain multple tokens, it is defined as a set in SMT. Structured token types defined in the HLPN are mapped to tuples in SMT, and primitive token types such as integer and strings are encoded as Integer in SMT.

Table I: HLPN Elements to SMT Sorts

HLPN Elements	SMT Sorts
Marking	Tuple
Places	Set
Structured Token Type	Tuple
Primitive Token Type	Integer

Since an execution sequence having k transition firing steps $M_0 \rightarrow M_1 \rightarrow \cdots \rightarrow M_k$ contains $k+1$ markings, which correspond to $k+1$ symbolic states in ϕ_k, $k+1$ sets of unique variables $\{V_0, V_1, \cdots, V_k\}$ in SMT are needed.

2) Encoding Initial State : In a HLPN model, the initial state is defined by the initial marking. In ϕ_k, an initial symbolic state is defined by assigning values to the first symbolic state through clauses. The clauses are mainly expressed in equations. Thus a formula representing the initial marking in the HLPN is first constructed and then added as a conjunct to ϕ_k.

3) Formulating Transitions: In a HLPN model, each transition t_j captures a local state change and its firing subtracts tokens from t_j's input places and adds tokens into t_j's output places, which contributes to the overall marking change $M_i \rightarrow M_{i+1}$. The effect of firing each transition is encoded as a logic formula $t_j\left(S_i, S_{i+1}\right)$. More specifically, let P be the set of places in HLPN model and p_{tj} be the set of places connected to t_j, the relation is encoded as Equation 2.

$$t_j\left(S_i, S_{i+1}\right) \quad = \quad \begin{cases} S_{i+1}\left(p_{tj}\right) = t_i\left(S_i\left(p_{tj}\right)\right) \\ S_{i+1}\left(P \setminus p_{tj}\right) = S_i\left(P \setminus p_{tj}\right) \end{cases} \tag{2}$$

Concurrent transition firings in the HLPN model need to be linearized, which does not affect the safety properties to be analyzed. Thus only interleaved executions are considered. Due to the non-determinism of transition firings, each firing (transition) step is encoded as a formula $T_i\left(S_i, S_{i+1}\right) = \bigvee_{j=0}^{n} t_j(S_i, S_{i+1})$ representing the disjunction of the formulas capturing the effects of firing individual transitions $t_j\ (0 \leq j \leq n)$. An execution consisting of k transition firings is formulated as a conjunction of k successive state transition formulas as follows, which is added as a conjunct to ϕ_k:

$$\bigwedge_{i=0}^{k-1} \left(\bigvee_{j=0}^{n} t_j(S_i, S_{i+1})\right) \tag{3}$$

4) Defining Property : In a HLPN model, properties are defined in FOLTL formula f. Since BMC is most effective in checking the violation of safety properties, a formula $f\left(S_i\right)$ representing the safety property formula f without temporal operators in state S_i needs to be checked. Formula $\bigvee_{i=0}^{k} \neg f(S_i)$ expressing the violation of the safety property in the first k transition step in an execution sequence is added as a conjunct to ϕ_k.

B. Specific Translation Code from HLPNs to Z3

PIPE+Verifier processes an HLPN model and translates it into C API code provided by Z3 solver so that Z3 solver can compile and execute the code.

The generated C API code contains five parts:

Table II: SMT Declaration Z3 Code

SMT Sort	Code Example	Relation to HLPN
StateTUPLE	Z3_mk_tuple_sort()	Marking
PlaceSetSORT	Z3_mk_set_sort()	Place
TokenSORT	Z3_mk_tuple_sort()	Structured token type
IntegerSORT	Z3_mk_int_sort()	Primitive token type

1) **Declaration**: declares a list of required types (called SORT in SMT), shown in Table II.
2) **Defining symbolic states**: the state builder defines $k+1$ states in C code, each state has a type STATETUPLE. The C code uses Z3_ast S_i = Z3_mk_const(STATE_TUPLE), where S_i is the identifier of state i.
3) **Building initial state**: since symbolic states are defined, a formula capturing the initial marking asserts that an empty place set equals to S_0. A code snippet is shown in Code 1:

Code 1: Initial State in Z3

```
Z3_ast ini_token_clauses[m];
  Z3_ast ini_place = Z3_mk_empty_set(TokenSORT);
  Z3_ast ini_token = Z3_mk_const(TokenSORT);
  Z3_mk_set_add(ini_place, ini_token);
  ...
  Z3_ast ini_token_clauses[0] =
      Z3_mk_eq(mk_unary_app(proj_decls[0], S0), ini_place);
  ...
Z3_assert_cnstr(ctx, Z3_mk_and(m, ini_token_clauses[0]));
```

4) **Formulating transitions**: the formula is defined in terms of a conjunction of k successive state transitions $T\left(S_i, S_{i+1}\right)$, where each state transition is defined by a disjunction of local transitions $t\left(S_i, S_{i+1}\right)$. $t\left(S_i, S_{i+1}\right)$ is defined by an $if - then - else$ structure $if\, c_0\, then\, c_1\, else\, c_2$, which is a concise representation of $(c_0 \implies c_t) \wedge (\neg c_0 \implies c_f)$. A code snippet is shown in Code 2:

Code 2: Transition Formulation in Z3

```
Z3_ast transitions_state[k];
   Z3_ast transitions_local_or[l];
        Z3_ast var_in = Z3_mk_const(Z3_mk_set_sort(TokenSORT)
              );
        ...
        Z3_ast cond_in = Z3_mk_set_member([input arc variable
              ], [input place]);
        Z3_ast cond_trans = [transition formula];
        Z3_ast cond_out = Z3_mk_set_member([output arc
              variable], [output place]);
        Z3_ast cond = Z3_mk_and(o, cond_and);
        Z3_ast trans_true_and[m];
        ...
        Z3_ast trans_true = Z3_mk_and(m, trans_true_and);
        Z3_ast trans_false_and[n];
        ...
        Z3_ast trans_false = Z3_mk_and(n, trans_false_and);
        Z3_ast transitions_local[0] = Z3_mk_ite(cond,
              trans_true, trans_false);
       ...
        Z3_ast trans_dump = Z3_mk_eq(S0, S1);
        Z3_ast transitions_local_dump = Z3_mk_implies(
              Z3_mk_true(), trans_dump);
        ...
  Z3_ast transitions_state[0] = Z3_mk_or(l,
        transitions_local_or);
  ...
Z3_assert_cnstr(ctx, Z3_mk_and(k, transitions_state));
```

5) **Defining properties**: each safety property f is defined by a disjunction of negated formulas in successive

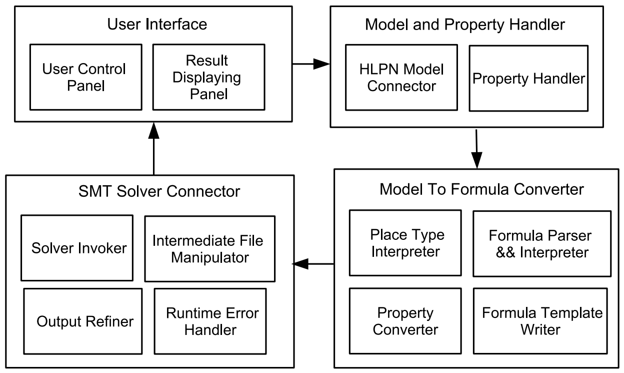

Figure 3: The Design View of PIPE+Verifier

states $\neg f\left(s_i\right)$. Thus a bad state indicated by a specific token reaching a particular place is checked using Z3_mk_set_member(). A code snippet is shown in Code 3:

Code 3: Property Definition in Z3

```
Z3_ast properties[k];
   Z3_ast token = Z3_mk_const(TokenSORT);
   ...
   property[0] = Z3_mk_set_member(token, mk_unary_app(place,
        S0));
...
Z3_assert_cnstr(Z3_mk_or(k, properties));
```

IV. PIPE+VERIFIER

PIPE+Verifier is developed as an additional analysis component of PIPE+ [12]. PIPE+ is a graphical HLPN editor and simulator. A user builds a HLPN model in PIPE+ by dragging and dropping graphical elements as well as editing specifications inside the graphical elements. PIPE+Verifier leverages this editor as an input source of HLPN models and launches PIPE+Verifier to conduct BMC on the models. Similar to PIPE+, PIPE+Verifier is implemented in Java, thus is able to run on any platform that can run Java Virtual Machine and Z3 solver. A detailed design view of PIPE+Verifier is shown in Figure 3:

1) User Interface: User interface in PIPE+Verifier is built as a dashboard (in Java Swing) that can take in user's input for properties in FOLTL and command to start the checking process. Furthermore, it displays the analysis results as well as error messages that may encounter during the checking process.

Properties are built in a standard format in order to conform to the safety property that BMC can check effectively. The format eases the transformation of FOLTL formula into SMT formula. In order to restrict the user's input format and specify the targeted bad state, a user is provided with a list of places fetched from the HLPN model and tokens need to be constructed according to the selected place types. A user needs to provide an upper bound k value required by BMC.

If a safety property holds for all states searched from the initial state to a depth up to predefined k, the displayer will show a message "SAT" and a resources consumed summary including time and memory usage; otherwise, a message "UNSAT" is shown and a counterexample leading to the bad state is printed.

2) Model and Property Handler: Model and property handlers are used to prepare for the next model to formula converting process. Because PIPE+Verifier uses HLPN model built from PIPE+ model editor, the model connector is built to refine the HLPN model from PIPE+ and check the consistency of the model to avoid conversion error. The property handler, on the other hand, takes the user input property from the interface and prepares its conversion to an SMT formula.

3) Model To Formula Converter: Model To Formula Converter is the component to conduct this conversion process. The converter consists of several components including State Builder, Place Type Recognizer, Transition Parser and Interpreter, Property Converter and Formula Template Writer.

- State builder: The state builder defines a STATETUPLE (shown in the SMT context above) according to the structure of the HLPN model. The tuple is a structure that consists of a list of place sorts, each place sort is also a tuple. The complete state list for the SMT formula in Equation 1 contains $k+1$ states.
- Place type recognizer: the recognizer traverses all the places in the HLPN model and stores all the distinct place types in order to construct distinct sorts in SMT context;
- Transition Parser and Interpreter: in the HLPN model, transition formula is a first-order logic formula that guards the token flow in the model. The formula needs to be parsed and interpreted into an abstract syntax tree in order to allow this tool to understand the first-order logic formula and build a corresponding SMT formula;
- Property Converter: in BMC, SMT solver's responsibility is to search for bad state, which is satisfiable solution to the negated properties. The safety properties prepared by property handler is converted into a negated SMT formula by Property Converter. The converting process is straightforward.
- Formula Template Writer: The template writer leverages a predefined template file in C language that contains necessary utility functions for checking the SMT formula in Z3 solver, and fills in model and property information in order to build the input file that conforms to Z3 solver's input format. Formula Template Writer writes the converted SMT formula's declarations, states, transitions, and properties into the file that can be compiled and checked by Z3 solver.

4) SMT Solver Connector: The SMT Solver Connector handles the process of delivering the SMT formula to Z3 solver as well as receiving the checking result from Z3 solver. The connector consists of four components:

- Solver Invoker: the invoker links to the tool to the backend engine Z3, which contains some scripts to automatically launch Z3 with proper parameters. The scripts are shell scripts for Windows and Unix in order to allow the tool to run on different platforms.
- Intermediate file manipulator: As the analysis process includes conversions and checkings, intermediate temporary files are created such as a file to represent formula,

a compiled Z3 checker, a file to record Z3's checking result, and a file to store final result. Intermediate file manipulator is in charge of the creating and deleting temporary files.

- Runtime error handler: Since PIPE+Verifier involves an external tool and file system interactions, unexpected errors can happen, an error handler can prevent the tool get into failure and can better managing errors.
- Output refiner: the raw result generated by Z3 solver is not readable as it only checks the satisfiability of the intermediate SMT formula instead of the original model and property. Thus, it is necessary to rebuild the model and property's checking result based on Z3 generated result. Output refiner can process the Z3 result file by removing redundant information and reorganize structure, and present readable results to the user.

V. An Improved Transition Formulation

The naive transition formulation given in the previous sections results in a ϕ_k capturing all the possible interleavings of transition firings in the given HLPN within depth k without considering the dependencies among them. The computation complexity of the naive method is thus exponential and is not computable with a large k value.

The firing of a transition depends on the existence of tokens from its input places P, thus depends on the other transitions producing tokens for P. If in a state s, a transition t's input places are empty or do not have enough tokens to enable t, t cannot fire at state s. If in a state s, a transition t's output places are not relevant to a given property, a transition firing sequence σ_k has a t as the last transition has no impact on the satisfiability of the property. With these observations and analysis, it is possible to build a more concise formula to avoid redundant checking by an SMT solver and thus improve the efficiency of BMC.

For example, in Figure 4, the initial marking is $P_0\,\{tok_0\}$, $P_1\,\{\}$, $P_2\,\{\}$, if we want to check whether it can reach a marking where $P_2\,\{tok_0\}$. The model formula produced by equation 1 with $k = 2$ is:

$$\phi_k = I(s_0) \wedge (t_i\,(s_0, s_1) \vee t_o\,(s_0, s_1)) \wedge \\ (t_i\,(s_1, s_2) \vee t_o\,(s_1, s_2)) \wedge (\neg f(s_0) \vee \neg f(s_1) \vee \neg f(s_2)) \quad (4)$$

This formula ϕ_k covers all possible transition firing orders including $t_i \rightarrow t_i$, $t_i \rightarrow t_o$, $t_o \rightarrow t_i$ and $t_o \rightarrow t_o$. There are infeasible firing sequences in this net model. Firing t_i twice cannot reach a marking in P_2 because P_2 is not directly updated by t_i. Firing t_o before t_i is impossible because P_1 is empty initially that cannot enable t_o if t_i has not yet fired. The only feasible firing sequence is $t_i \rightarrow t_o$. We can build a reduced formula ϕ':

$$\phi' = I(s_0) \wedge (t_i\,(s_0, s_{temp}) \wedge t_o\,(s_{temp}, s_2)) \\ \wedge (\neg f(s_0) \vee \neg f(s_1)) \quad (5)$$

where s_{temp} is an intermediate state for a consecutive firings of t_i and t_o, and does not need to be checked.

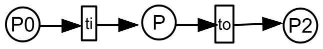

Figure 4: A Simple Model

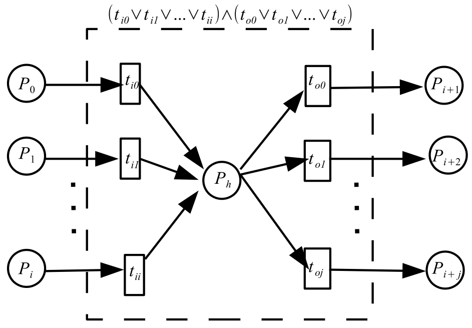

Figure 5: A Structural Pattern

A. A New Structural Pattern

The above observations show that exploring transition dependencies can field much concise formulas that can be solved more efficiently. It is well known that many simple structural reductions such as removing self-loop can be done to a given Petri net to obtain a behavioral equivalent yet simpler Petri nets. However simple net structural transformation rules need to be applied carefully with regard to high level Petri nets since the removed net elements may contain critical information such type information in places, arc label expressions, and constraints associated with transitions, as a result the reduced net may not be behavioral equivalent to the original net. We have developed the following general structural and conceptual (only used during formula translation) reduction rule, and proved its correctness - behavioral preservation.

Figure 5 shows a place P_p that is connected by a set of transitions $T_p = \{t_{i0}, t_{i1}, \ldots, t_{iu}, t_{o0}, t_{o1}, \ldots, t_{ov}\}$. P_p's input transition set is $T_{pi} = \{t_{i0}, t_{i1}, \ldots, t_{iu}\}$and output transition set is $T_{po} = \{t_{o0}, t_{o1}, \ldots, t_{ov}\}$.

Under the following conditions:

1) All the arc label connected to P_p are simple variables;
2) P_p is neither an initial marking place nor a property identified place;
3) P_p is the only output place of all T_{pi} and the only input place of all T_{po}.

Let s' be a successor state of s and s'' be a successor state of s'. A new and much more concise subformula (equation 6) of ϕ_k is obtained:

$$T_p(s, s'') = (t_{i0}\,(s, s') \vee t_{i1}\,(s, s') \vee ... \vee t_{iu}\,(s, s')) \wedge \\ (t_{o0}\,(s', s'') \vee t_{o1}\,(s', s'') \vee ... \vee t_{ov}\,(s', s'')) \quad (6)$$

B. Experiment Results For Refined Transition Formula Construction

Figure 6 shows a share memory model in HLPN. In this model, the pattern can be applied to place $P_{OwnMemAcc}$'s input transition $T_{Begin\_Own\_Acc}$ and output transition $T_{End\_Own\_Acc}$, thus the pattern is defined as

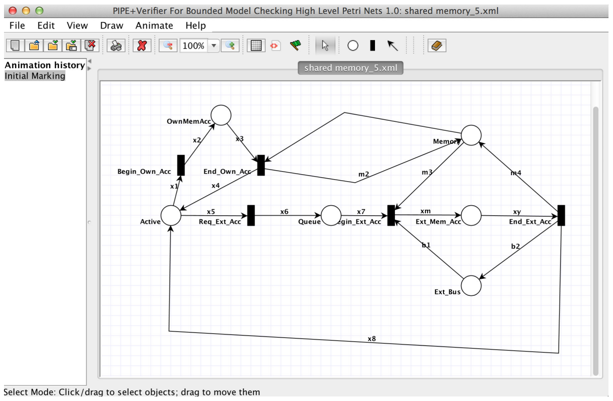

Figure 6: Shared Memory Model in PIPE+Verifier

Table III: Performances of Checking Shared Memory Model using Old and Refined Methods

Procs Number	**Bound** Step	**Time** Old	**Time** Refined	**Heap** Old	**Heap** Refined
5	5	0.07s	0.05s	0.86mb	0.78mb
5	10	0.30s	0.23s	1.54mb	1.34mb
5	15	1.49s	1.20s	2.53mb	2.42mb
10	5	0.12s	0.10s	1.02mb	1.00mb
10	10	0.98s	0.84s	2.08mb	1.97mb
10	15	15.50s	8.37s	4.73mb	4.60mb

$T_{Begin\_Own\_Acc} \wedge T_{End\_Own\_Acc}$. Table III presents a comparison of time and memory consumption of the naive transition construction method and the refined construction method. Despite the pattern is applied once in this model, some performance improvements are clear as shown in Equation 7:

$$\Box\neg(marking\,(Ext\_Mem\_Acc) = \langle 3, 0\rangle \\ \wedge\, marking\,(Ext\_Mem\_Acc) = \langle 2, 4\rangle) \quad (7)$$

VI. Conclusion

In this paper, we presented a tool called PIPE+Verifier that supports BMC safety properties of HLPN models, which has been successfully applied to analyze a variety of HLPN net models [13]. The tool automatically converts a HLPN model into a SMT formula, leverages a SMT solver Z3 to solve the formula, and then displays the checking results. We provided both functional and design views of this tool, which facilitate advanced users to extend this open source tool (https://github.com/liusu1011/PIPE-Verifier.git) easily. We discussed additional improvements that can make PIPE+Verifier more efficient. We are working on several other potential extensions to make PIPE+Verifier more efficient and powerful. Currently, the transitions are formulated in a breadth first approach based on the original BMC idea [5], thus a SMT solver explores all possible transition sequences of one step, two steps, and up to $k$steps. Many of these transition sequences may not be relevant to the property being checked. An alternative depth first formulation chaining k transition firings utilizing structural transition dependencies as well as the checked property may be solved much more efficiently. PIPE+Verifier currently only supports BMC of safety properties. Integrating a SMT solver with induction techniques [14] can become a full fledged analysis methodology to check safety properties completely.

Acknowledgments We thank three anonymous reviewers for their helpful comments. This work was partially supported by the NSF under grant HRD-0833093 and by the AFRL under agreement number FA8750-15-2-0106. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] Cpn tools. http://cpntools.org.
[2] *High-level Petri Nets - Concepts, Definitions and Graphical Notation*, 2000.
[3] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using smt solvers instead of sat solvers. *Int. J. Softw. Tools Technol. Transf.*, 11(1):69–83, January 2009.
[4] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.
[5] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.
[6] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. In *Formal Methods in System Design*, page 2001. Kluwer Academic Publishers, 2001.
[7] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011.
[8] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.
[9] Steve Hostettler, Alexis Marechal, Alban Linard, Matteo Risoldi, and Didier Buchs. High-level petri net model checking with alpina. *Fundamenta Informaticae*, 113(3-4):229–264, August 2011.
[10] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *Int. J. Softw. Tools Technol. Transf.*, 9(3):213–254, May 2007.
[11] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.
[12] Su Liu, Reng Zeng, and Xudong He. PIPE+ - A modeling tool for high level petri nets. In *Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering (SEKE'2011), Eden Roc Renaissance, Miami Beach, USA, July 7-9, 2011*, pages 115–121, 2011.
[13] Su Liu, Reng Zeng, Zhuo Sun, and Xudong He. Bounded model checking high level petri nets in pipe+verifier. In *Formal Methods and Software Engineering - 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings*, pages 348–363, 2014.
[14] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, FMCAD '00, pages 108–125, London, UK, UK, 2000. Springer-Verlag.