Bounded Model Checking High Level Petri Nets in PIPE+Verifier

Su Liu, Reng Zeng, Zhuo Sun, and Xudong He

Florida International University, Miami, Florida 33199, USA
{sliu002,rzeng001,zsun003,hex}@cis.fiu.edu

Abstract. High level Petri nets (HLPNs) have been widely applied to model concurrent and distributed systems in computer science and many other engineering disciplines. However, due to the expressive power of HLPNs, they are more difficult to analyze. Exhaustive analysis methods such as traditional model checking based on fixed point calculation of state space may not work for HLPNs due to the state explosion problem. Bounded model checking (BMC) using satisfiability solvers is a promising analysis method that can handle a much larger state space than traditional model checking method. In this paper, we present an analysis method for HLPNs by leveraging the BMC technique with a state-of-the-art satisfiability modulo theories (SMT) solver Z3. A HLPN model and some safety properties are translated into a first order logic formula that is checked by Z3. This analysis method has been implemented in a tool called PIPE+Verifier and is completely automatic. We show our results of applying PIPE+Verifier to several models from the Model Checking Contest @ Petri Nets and a few other sources.

Keywords: Formal Methods, Petri Nets, Model Checking, Bounded Model Checking.

1 Introduction

Petri nets are a graphical formal language to model concurrent and distributed systems. Low level Petri nets are suitable to model control flows but cannot effectively model data and functionality in complex systems. High level Petri nets (HLPNs) [2] are a more expressive formalism developed to handle data and functionality in addition to control flows.

HLPNs are executable. Tools like CPN Tools [25] and PIPE+ [30] support the modeling and execution of different forms of HLPNs. However, analysis by simulation can only explore a finite number of executions and thus cannot assure safety properties to be satisfied in all possible executions. Traditional model checking [26] is an automatic and exhaustive analysis method to explore all possible executions of a model, but suffers from the state explosion problem. Bounded model checking (BMC) with satisfiability solving [9,13] was proposed as an alternative approach to address the state explosion problem in the traditional model checking approach. In BMC, a feasible symbolic execution of a transition

© Springer International Publishing Switzerland 2014

system and the negation of some safety property are translated into a logic formula, which is checked by a satisfiability solver. If the formula is satisfiable, a counter example is found and thus the safety property does not hold. On the other hand, if the formula is not satisfiable up to a pre-defined upper bound k, the safety property holds up to k. Although this approach is not a complete technique for safety property analysis, it has been shown to be very effective in detecting the violation of safety properties in many real-world applications.

Encoding a low level Petri net model into a propositional logic formula is straightforward, but encoding a HLPN model is not since HLPNs use structured data and algebraic expressions to define functionality. In recent years, great progress has been made on satisfiability modulo theories (SMT) [16,32] solvers that can check the satisfiability of a subset of first-order logic formulas with a variety of underlying theories including linear arithmetic, difference arithmetic and arrays. These SMT solvers are expressive enough to represent the data and algebraic expressions in HLPNs naturally. Furthermore, SMT solvers are becoming more efficient according to the annual competitions results from SMT [7], and have been successfully integrated into verification tools such as CBMC [3], SLAM2 [5], and VS3 [33].

In this paper, we present a method for using SMT solvers to perform bounded model checking on HLPNs. We leverage the theory of sets [29] that has been integrated to some SMT solvers to represent HLPNs, where a place can have zero or more tokens. Similar to BMC, our method specifies a k value before checking, which defines the upper bound of transition firing actions (state changes). For each safety property violated within k steps, a transition firing sequence leading to an error state is generated. However, this method is incomplete because the upper bound k is often not given in real applications. Reference [15] discussed the complexity of finding a complete threshold.

We have implemented a prototype tool called PIPE+Verifier, which integrates the state of the art SMT solver Z3 [17]. We have applied PIPE+Verifier to analyze several models from Model Checking Contest @ Petri Net [27], a Mondex model [43] (an electronic purse system proposed as the first pilot project in the worldwide formal verification grand challenge) and a model given in [38,37]. We have provided a comparison of our tool with related Petri nets tools and symbolic model checking tools.

2 High Level Petri Nets

A HLPN graph [2] comprises: a net graph, place types, place markings, arc annotations, transition conditions, and declarations. The net graph is a 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). A place type is a power set of tokens. A token type can be a tuple of primitive data types such as integer and string. 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 Boolean expressions. Declarations comprise definitions of place types, variable types, and functions.

A HLPN is executable. A transition is enabled if its input places have the right tokens in the current marking that satisfy the transition condition. An enabled transition can fire and result in a new marking by subtracting the tokens from the input places and adding new tokens to the output places according to the corresponding arc annotations. Multiple enabled non-conflict transitions may fire simultaneously. An execution of a HLPN is a sequence of transition firings from the given initial marking. The behavior of a HLPN is the set of all possible executions.

Figure 1 illustrates a dining philosopher problem modeled in 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 $P_{Chopsticks}$ are both initiated with markings that have five tokens $\{\langle 0\rangle,\langle 1\rangle,\langle 2\rangle, \langle 3\rangle,\langle 4\rangle\}$. T_{Pickup}'s transition condition is $p = c_1 \wedge (p+1)\%5 = c_2 \wedge e = p$. $T_{Release}$'s transition condition is $p = r \wedge c_1 = r \wedge c_2 = (r+1)\%5$.

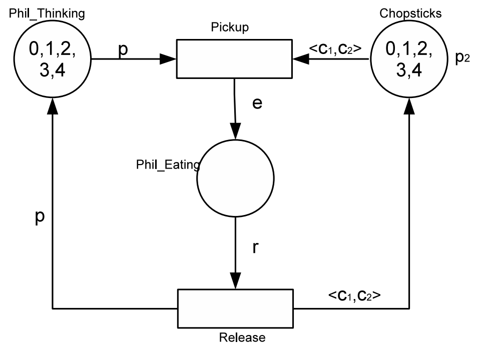

Fig. 1. 5-Dining Philosophers Problem in High Level Petri Net

3 Satisfiability Modulo Theories Solvers

Satisfiability modulo theories (SMT) [16] support a combination of 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.

3.1 Important Theories

Some important high level theories supported by SMT solvers are listed below as the foundation of our method.

Arrays. The theory of arrays [35,4] in SMT solvers is different from the ones in standard programming languages. In SMT, an array's size can be infinite. There are two built in functions: $select : ARRAY \times INDEX \rightarrow ELEM$ and $store : ARRAY \times INDEX \times ELEM \rightarrow ARRAY$ where $ARRAY$, $INDEX$, $ELEM$ are the sorts of the array, the index of the array and the elements in the array.

Tuples. The theory of tuples [29] supports a data structure with a list of components and access to individual components by projection.

Sets. A set is a collection of objects. Reference [29] has defined a set theory, which has been implemented in several SMT solvers [8]. The theory of sets in SMT solvers supports a list of set operations including set member $\in$, set subset $\subseteq$, set union $\cup$, set intersect $\cap$ and set difference $\setminus$.

3.2 Z3

In recent years, the efficiency of SMT solvers has been greatly improved. An annual SMT competition is held every year [8] and the participants include CVC4 [6], Z3 [17], MathSAT [12], Opensmt [10], and Yices [19]. Among them, Z3 [17], developed by Microsoft Research Institution, is reported to have the largest number of users and supports almost all the popular SMT background theories such as rational and integer arithmetic, bit-vectors, array theory, and set theory. In addition, Z3 has been adopted as the backend verification engine for a variety of tools, such as VS3 [33], SLAM2 [5] and CBMC [3]. Z3's developing team provides api and documentation for different programming languages (C, C++, .NET, Python). Therefore, we have selected and integrated Z3 into our tool as the backend satisfiability solving engine.

4 Bounded Model Checking High Level Petri Nets

Given a finite transition system M, a LTL formula f and an integer k, existential bounded model checking (BMC) [9] tries to determine whether there exists a computation path in M of at most length k (denoted as M_k) that satisfies f. To realize BMC, a logic formula ϕ_k from M, f and k is constructed and checked using a constraint solver. ϕ_k is satisfiable if and only if there is a path p of length at most k in M_k that satisfies f. The satisfying assignment for ϕ_k is called a witness for path p. However, BMC is in general not able to determine the satisfiability of a formula f since k's upper bound is unknown in many real-world applications. [15] shows that finding the upper bound for k is as complex as traditional model checking. To check the validity of a safety property up to k steps using existential BMC, we use f to represent the negated safety property. Thus the safety property holds as long as f is not satisfiable. [9] shows BMC can check all formulas in ACTL* [20].

In the following sections, we present a translation schema of applying bounded model checking to HLPNs.

4.1 General Idea of BMC using SMT Solver

In BMC, a logic formula ϕ_k is constructed from a given M_k, including the initial state I and unrolled transition relations T, and some negated safety properties f. Since 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:

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

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 negated safety property in unrolled state s_i $(0 \leq i \leq k)$. 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 $(0 \leq i \leq k)$ that satisfies f, thus violates the safety property. Otherwise, the safety property holds in M within k transition firings.

The general SMT logic context for BMC is shown in Figure 2:

DEF

s : STATETUPLE

ASSERT

$$\begin{aligned} & \mathit{Initial\_marking}(s_0) \\ \wedge & \bigwedge_{i=0}^{k-1} \mathit{Transition}(s_i, s_{i+1}) \\ \wedge & \bigvee_{i=0}^{k} \mathit{Negated\_property}(s_i) \end{aligned}$$

CHECK

Fig. 2. SMT context for bounded model checking

4.2 Represent HLPNs in SMT Context

Our goal is to translate a given HLPN model to a logic formula shown in Figure 2, and then use an SMT solver to check its satisfiability.

Define States in SMT Context. In HLPNs, a state s_i is defined by a marking that is a distribution of tokens in places. Each place can contain 0 or more tokens (the number may be bounded or unbounded) and tokens can be structured data. To define a state in SMT context, a hierarchical layered data structure is constructed.

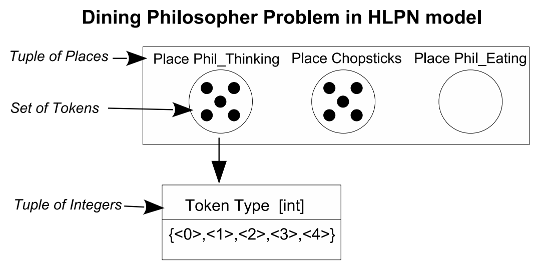

Fig. 3. An inner view of dining philosophers problem in HLPN model

A state s_i is defined by a tuple whose elements are places: $s_i \doteq \langle p_0, p_1, \ldots, p_n \rangle$. Each place p_j $(0 \leq j \leq n)$ is defined by a set containing $m \geq 0$ tokens: $p_j \doteq \{tok_0, tok_1, \ldots, tok_m\}$. Each token tok_k $(0 \leq k \leq m)$ is defined by a tuple of primitive data elements: $tok_k \doteq \langle e_0, e_1, \ldots, e_l \rangle$. Figure 3 shows an inner view of a HLPN model. In Figure 3, the tuple of places is $\langle P_{Phil\_Thinking}, P_{Chopsticks}, P_{Phil\_Eating} \rangle$, in which place $P_{Phil\_Thinking}$ has 5 tokens $\{\langle 0 \rangle, \langle 1 \rangle, \langle 2 \rangle, \langle 3 \rangle, \langle 4 \rangle\}$ and each token tok_k has only one field $\langle ID \rangle$ whose type is Integer.

In the SMT context, a state is defined by type STATETUPLE. The hierarchical data structure that constitutes STATETUPLE is shown in Table 1.

Table 1. High level Petri net elements mapped to SMT theory

HLPN Elements	**SMT Theory**	**In PIPE+Verifier**
HLPN Model	Tuple (Places)	STATETUPLE
Place Type	Set (Tokens)	SET*i*SORT
Token Type	Tuple (Integer or String Values)	DT*i*SORT
Primitive Data	Integer or String	INTSORT

Define the Initial State. The *Inital_marking* (s_0) in Figure 2 is defined from the initial marking M_0 of a HLPN model. The state s_0 contains tokens of all the places marked in M_0.

Define Transitions in SMT Context. $Transition(s_i, s_{i+1})$ in Figure 2 is a binary relation between the current state s_i and the next state s_{i+1}. In BMC, the upper bound of the transition firing sequence is k, thus the state transition of ϕ_k is unrolled k times, denoted as $\bigwedge_{i=0}^{k-1} \mathit{Transition}(s_i, s_{i+1})$. A HLPN model consists of $n \geq 0$ transitions $t_0, t_1, \ldots, t_n$, and any one of them may fire if enabled, thus $\mathit{Transition}(s_i, s_{i+1})$ is represented by a disjunction of the transitions in the HLPN model $\bigvee_{j=0}^{n} t_j(s_i, s_{i+1})$. Transitions in ϕ_k are defined as the formula shown in Equation 2:

$$\bigwedge_{i=0}^{k-1} (\mathit{Transition}(s_i, s_{i+1})) = \bigwedge_{i=0}^{k-1} (\bigvee_{j=0}^{n} t_j(s_i, s_{i+1})) \tag{2}$$

Each transition in the HLPN model $t_j(s_i, s_{i+1})$ with a precondition (captured by c_0) and a post-condition (captured by c_1) are defined in an if-then-else structure *if* c_0 *then* c_1 *else* c_2 , representing $(c_0 \implies c_1) \wedge (\neg c_0 \implies c_2)$. The translation schema is described below:

- If condition c_0:
  - Use set membership operation to check if each input place in s_i has at least one token;
  - In state s_i, each transition condition clause corresponds to a constraint;
- Case True c_1:
  - Tokens are removed from t_j's input places of state s_i using set difference operation;
  - New tokens are added to t_j's output places of state s_{i+1};
  - Tokens in unrelated places in state s_i remain the same in those places in s_{i+1};
- Case False c_2: tokens in all places in the next state s_{i+1} are the same as in the current state s_i.

Define Properties in SMT Context. To check a safety property, we define $\mathit{Negated\_property}(s_i)$ as the negation of the safety property. If there exists a state s_i satisfies $\mathit{Negated\_property}(s_i)$, the safety property is violated at s_i. Thus, a disjunction of $\mathit{Negated\_property}(s_i)$ $0 \leq i \leq k$ is asserted in ϕ_k.

A Translation Example – Dining Philosophers Problem. From the dining philosophers HLPN given in Figure 1, we obtain the following translation:

1. State Definition: As shown in Figure 4, a state, consists of three places, are defined as three sets in STATETUPLE. All of the sets have the same set type $DTSORT$, and their element types are $INSORT$.
2. Initial state: place $P_{Phil\_Thinking}$ set contains five philosophers whose IDs are $\{\langle 0\rangle, \langle 1\rangle, \langle 2\rangle, \langle 3\rangle, \langle 4\rangle\}$ and place $P_{Chopsticks}$ has five chopsticks whose IDs are $\{\langle 0\rangle, \langle 1\rangle, \langle 2\rangle, \langle 3\rangle, \langle 4\rangle\}$. Therefore, as shown in Figure 5, both places at state s_0 contain five tokens.

DEF.

$$\begin{aligned} \mathit{STATETUPLE} &\equiv \langle P_{\mathit{Phil\_Thinking}} : \mathit{SETSORT}, \\ &\qquad P_{\mathit{Chopsticks}} : \mathit{SETSORT}, \\ &\qquad P_{\mathit{Phil\_Eating}} : \mathit{SETSORT} \rangle \\ \mathit{SETSORT} &\equiv \{\mathit{set} : \mathit{DTSORT}\} \\ \mathit{DTSORT} &\equiv \{\mathit{int} : \mathit{INTSORT}\} \\ \mathit{State} &\equiv \{s_0 : \mathit{STATETUPLE} \\ &\qquad s_1 : \mathit{STATETUPLE} \\ &\qquad \ldots \\ &\qquad s_k : \mathit{STATETUPLE} \} \end{aligned}$$

Fig. 4. State definitions of 5-dining philosophers in SMT logic

$$\begin{aligned} \mathit{Initial\_marking}(s_0) \;\equiv\; & P_{\mathit{Phil\_Thinking}}(s_0) = \{\langle 0\rangle, \langle 1\rangle, \langle 2\rangle, \langle 3\rangle, \langle 4\rangle\} \\ & \wedge P_{\mathit{Chopsticks}}(s_0) = \{\langle 0\rangle, \langle 1\rangle, \langle 2\rangle, \langle 3\rangle, \langle 4\rangle\} \\ & \wedge P_{\mathit{Phil\_Eating}}(s_0) = \emptyset \end{aligned}$$

Fig. 5. Initial State of 5-Dining Philosopher in SMT Logic

$$\begin{aligned} \bigwedge_{i=0}^{k-1} \mathit{Transition}(s_i, s_{i+1}) \;\equiv\; & (T_{\mathit{Pickup}}(s_0, s_1) \vee T_{\mathit{Release}}(s_0, s_1)) \\ & \wedge (T_{\mathit{Pickup}}(s_1, s_2) \vee T_{\mathit{Release}}(s_1, s_2)) \\ & \ldots \\ & \wedge (T_{\mathit{Pickup}}(s_{k-1}, s_k) \vee T_{\mathit{Release}}(s_{k-1}, s_k)) \end{aligned}$$

```
$T_{Pickup}(s, s') \equiv$
  IF   $p \in P_{Phil\_Thinking}$
   $\wedge\ l \in P_{Chopsticks}$
   $\wedge\ r \in P_{Chopsticks}$
   $\wedge\ p = l \wedge (p+1)\%5 = r$
  THEN
  $P'_{Phil\_Thinking} = P_{Phil\_Thinking} - \{p\}$
   $\wedge\ P'_{Chopsticks} = P_{Chopsticks} - \{l\} - \{r\}$
   $\wedge\ P'_{Phil\_Eating} = P_{Phil\_Eating} \cup \{p\}$
  ELSE
  $P'_{Phil\_Thinking} = P_{Phil\_Thinking}$
   $\wedge\ P'_{Chopsticks} = P_{Chopsticks}$
   $\wedge\ P'_{Ehil\_Eating} = P_{Phil\_Eating}$
```

```
$T_{Release}(s, s') \equiv$
  IF   $p \in P_{Phil\_Eating}$
  THEN
  $P'_{Phil\_Thinking} = P_{Phil\_Thinking} + \{p\}$
   $\wedge\ P'_{Chopsticks} = P_{Chopsticks} \cup \{p\}$
                                   $\cup \{(p+1)\%5\}$
  ELSE
  $P'_{Phil\_Thinking} = P_{Phil\_Thinking}$
   $\wedge\ P'_{Chopsticks} = P_{Chopsticks}$
   $\wedge\ P'_{Phil\_Eating} = P_{Phil\_Eating}$
```

Fig. 6. State Transition of 5-Dining Philosophers in SMT Logic

$$\bigvee_{i=0}^{k} \mathit{Negated\_property}(s_i) \equiv (f(s_0) \vee f(s_1) \vee ... \vee f(s_k))$$

$$f \equiv P_{\mathit{Phil\_Eating}} = \{\langle 0\rangle, \langle 1\rangle\}$$

Fig. 7. Property Definition of 5-Dining Philosophers in SMT Logic

3. State transition: *Transition* is defined as $k-1$ transition steps that constrain pairs of consecutive states. Each transition step is an if-then-else structure that captures the pre-condition and post-condition of every local transition in HLPN. In Figure 6, s indicates the current state and s' indicates the next state.
4. Property definition: negated property $f\ (s_i)$ is state based, we need to define k disjunctions of error states. If one of $f\ (s_i)$ evaluates true, the whole formula is satisfiable and an error state s_i is reached. Figure 7 defines a simple negated safety property that the neighboring philosophers with ID $\{\langle 0\rangle, \langle 1\rangle\}$ can eat at the same time.

5 Evaluation

We have implemented an automated prototype tool called PIPE+Verifier to support our method and applied it to check relevant safety (reachability) properties in several benchmark problems modeled in HLPN. All experiments were conducted on a 32-bit Intel Core Duo CPU @3.0GHz box, with 4GB of RAM, running 32-bit Ubuntu.

5.1 Selected Benchmark Problems from Model Checking Contest @ Petri Nets

Model Checking Contest @ Petri nets (MCC) [27,28] is held annually to assess Petri nets based formal verification tools and techniques. Petri net verification tools are compared with regard to the scaling abilities, efficiency, and property checking capabilities on selected benchmark problems. The benchmark problems are modeled in low level Petri nets and Colored Petri nets. However, none of the participating tools produced any promising results on checking colored Petri net models. We have translated several Colored Petri net models into PIPE+Verifier and analyzed their safety (reachability) properties. We have examined the scalability of our tool by changing parameters in the model and varying bound k. The running results are presented below.

Dining Philosophers Model. In the previous section, we presented the 5-dining philosophers model. We have selected the following two negated safety properties to check in PIPE+Verifier.

$$\Box\neg\,(\mathit{marking}\,(\mathit{Phil\_Eating}) = 4 \wedge \mathit{marking}\,(\mathit{Phil\_Eating}) = 3) \tag{3}$$

$$\Box\neg(\mathit{marking}\,(\mathit{Phil\_Eating}) \neq 4 \wedge \mathit{marking}(\mathit{Phil\_Eating}) = 1$$
$$\wedge \mathit{marking}\,(\mathit{Chopsticks}) \neq 4) \tag{4}$$

The scaling parameter is the number (up to 20) of philosophers. The experiment results are shown in Table 2. For property 3, the PIPE+Verifier did not return a result when bound k reached 15 due to the exponential growth of the search space of Z3.

Table 2. Verifying Dining Philosophers Model

Philosophers	**Formula**	**Step Bound**	**Verdict**	**Property Hold**	**Time (seconds)**	**Heap Size (Mb)**
5	(3)	5	unsat	yes	0.41	1.72
5	(3)	10	unsat	yes	79.93	9.97
5	(3)	15	N/A	N/A	N/A	N/A
5	(4)	2	sat	no	0.25	1.25
10	(4)	2	sat	no	0.76	1.62
20	(4)	2	sat	no	3.23	2.63

Shared Memory Model. In [11], a shared memory model involving P processors was given. These processors can access their local memories as well as compete for shared global memory using a shared bus. We have built a HLPN model based on the above shared memory model and checked the following two negated safety properties:

$$\Box\neg(\mathit{marking}\,(\mathit{Ext\_Mem\_Acc}) = \langle 1,5\rangle \wedge \mathit{marking}\,(\mathit{Ext\_Bus}) = 1) \tag{5}$$

$$\Box\neg(\mathit{marking}\,(\mathit{Ext\_Mem\_Acc}) = \langle 1,5\rangle \wedge \mathit{marking}\,(\mathit{Memory}) \neq 4) \tag{6}$$

The scaling parameter is the number (up to 20) of processors P. The results are shown in Table 3.

Token Ring. A token ring [18] model shows a system with a set of M machines connected in a ring topology. Each machine can determine if it has the privilege (the right) to perform an operation based on its state and its left neighbor.

We have modeled a token ring using HLPN and selected the following two negated safety properties to check:

$$\Box\neg(\mathit{marking}\,(\mathit{State}) = \langle 3,0\rangle \wedge \mathit{marking}\,(\mathit{State}) = \langle 2,4\rangle) \tag{7}$$

$$\Box\neg(\mathit{marking}\,(\mathit{State}) = \langle 3,0\rangle \vee \mathit{marking}\,(\mathit{State}) = \langle 2,4\rangle) \tag{8}$$

The scaling parameter is the number of machines M, which is up to 20. The results are shown in Table 4.

Table 3. Verifying Shared Memory Model

Processors	**Formula**	**Step Bound**	**Verdict**	**Property Hold**	**Time (seconds)**	**Heap Size (Mb)**
5	(5)	5	unsat	yes	0.07	0.86
5	(5)	10	unsat	yes	0.3	1.54
5	(5)	15	unsat	yes	1.49	2.53
5	(6)	3	sat	no	0.75	1.80
10	(6)	3	sat	no	1.3	2.09
20	(6)	3	sat	no	13.05	4.35

Table 4. Verifying Token Ring Model

Machines	**Formula**	**Step Bound**	**Verdict**	**Property Hold**	**Time (seconds)**	**Heap Size (Mb)**
5	(7)	5	unsat	yes	0.32	1.34
5	(7)	10	unsat	yes	24.12	5.56
5	(7)	15	N/A	N/A	N/A	N/A
5	(8)	3	sat	no	0.09	1.01
10	(8)	3	sat	no	0.21	1.34
20	(8)	3	sat	no	0.86	2.03

5.2 Mondex

Mondex [43] smart card system is an electronic purse payment system, which involves a number of electronic purses with values and can exchange the values through a communication device. Mondex was the first pilot project of the International Grand Challenge on Verified Software [40], and was awarded the highest assurance level of secure systems, ITSEC Level E6 [41].

Mondex was first formally specified and proved using Z language [34]. Our previous work [43,42] formalized Mondex abstract and concrete models using HLPN. The concrete model depicts a transaction through nine operations {startFrom, startTo, readExceptionLog, req, ask, val, exceptionLogResult, exceptionLogClear, forged} and four status $\{idle, epr, epv, epa\}$. In this work, we have modeled the Mondex using PIPE+ [30] and verified a property "No Value Created" [42,41].

The HLPN model is initialized with two purses and one transaction proposal message. The safety property specifies that the sum of all the purses' balances does not increase: $\Box\ purse1.balance + purse2.balance \leq balance\_sum$. Since nine transitions may be involved in this transaction process, we set $k = 9$. Our model-checking result shows this transaction process is preserved since the negation of the safety property defined by f is not reachable in $k = 9$ transition firing steps. The time and memory consumed for this checking process are 27.85s and 11.42 Mbytes respectively.

5.3 Abstract State Machine Model

In [38], a method for checking symbolic bounded reachability of abstract state machines was presented. An abstract state machine written in AsmL was translated into a logic formula checked by an SMT solver with rich background theories including set comprehensions. The running times of the prototype tool in [38] and our tool PIPE+ Verifier on property *Count*(n) are shown in Table 5.

Table 5. Running time of checking Count model

Model program	Step bound	Verdict	Time of M.Veanes's Tool	Time of PIPE+Verifier
Count(5)	10	Sat	0.14s	1.43s
Count(5)	9	Unsat	1.5s	0.24s
Count(8)	16	Sat	2.2s	86.1s
Count(8)	15	Unsat	152s	15.26s

6 Related Work

6.1 Petri Nets Tools

Model checking Petri nets continues to be an active research topic. Various tools for modeling and verifying various forms of Petri nets have been built. Some of them are no longer maintained due to the evolution of new techniques. A Petri net model checking contest is held annually for the evaluation of some active tools. Table 6 lists the most recent participating tools (except for the last two). In this table, ALPiNA [23], Neco [21], CPN Tools [25] and SAMAT [31] support different types of high level Petri nets.

Colored Petri Nets Tool. Colored Petri Nets (CPNs) [25] are a kind of high level Petri nets that use tokens with typed values and functional programming language Standard ML [36] to define the guards of transitions. CPN Tools [1] is an industrial strength tool that is widely used to analyze modeled systems through simulation and model checking. CPN Tools integrates a model checking engine that explicitly searches the whole state space of a model.

ALPiNA. ALPiNA [23] is a model checker for algebraic Petri nets (APNs), which use algebraic abstract structured data type (AADTs) to define data and term equations to define transition guards and arc expressions. To symbolically model checking APNs, ALPiNA uses extended binary decision diagrams (BDDs) to represent the state space.

Table 6. Analysis Tools for Petri Nets

Name	Petri Net Type	Model Checking Technique
ALPiNA	Algebraic Petri Nets	Decision Diagrams
Cunf	Contextual Net	Net Unfolding, Satisfiability Solving
GreatSPN	Stochastic Petri Nets	Decision Diagrams
ITS-Tools	(Time) Petri Nets, ETF, DVE, GAL	Decision Diagrams, Structural Reductions
LoLA	Place/Transition Nets	Explicit Model Checking, State Compression, Stubborn Sets
Marcie	Stochastic Petri Nets	Decision Diagrams
Neco	High Level Petri Nets	Explicit Model Checking
PNXDD	Place/Transition Nets	Net Unfolding, Decision Diagrams, Topological
Sara	Place/Transition Nets	Satisfiability Solving, Stubborn Sets, Topological
CPN Tools	Colored Petri Nets	Explicit Model Checking
SAMAT	High Level Petri Nets	Explicit Model Checking

Neco. Neco [21] is a Unix toolkit that checks the reachability and other properties of high level Petri nets. Neco supports high level Petri nets annotated with Python objects and Python expressions. For model checking, Neco explicitly builds the state space.

SAMAT. SAMAT [31] is a tool for modeling and analyzing software architecture descriptions where component behavior models are expressed in predicate transition nets. SAMAT leverages an existing on-the-fly model checker SPIN [22] to check the satisfiability of properties expressed in linear time temporal logic in predicate transition net models.

6.2 Symbolic Model Checking Tools

Alloy. Alloy analyzer [24] is a software tool for analyzing a system defined in the Alloy specification language. The analysis in Alloy is based on reducing a model to a propositional formula and leveraging a SAT solver to solve the formula.

Java Path Finder. JPF [39] is a verification and testing environment for Java that integrates techniques such as model checking, program analysis and testing. Despite its state compression technique, JPF still cannot avoid the state explosion problem especially in terms of memory and time in checking high level data structures such as array.

CBMC and SMT-CBMC. C Bounded Model Checker (CBMC) [14] is an SAT based bounded model checker on C programs. SMT-CBMC [3] is an SMT based model checker that has significant improvement over the traditional SAT based model checkers. SMT-CBMC encodes sequential C programs into more compact first-order logic formulas that can be solved by SMT solvers.

7 Conclusion

In this paper, we have presented a method to analyze safety properties of HLPNs. Our method translates a HLPN model along with the negation of safety properties into a first order logic formula and uses the state of the art SMT solver Z3 to solve this formula. Our method is sound but incomplete since it requires a k value as an upper-bound to limit the length of firing sequences. By leveraging the theory of set in SMT solvers, our method supports HLPNs with unlimited number of tokens. However, checking a model with a large number of tokens may lead to an explosion of checking time. We have implemented this analysis method into a prototype, PIPE+Verifier, and embedded it into PIPE+, a graphical HLPNs modeling and simulation tool [30]. PIPE+Verifier is capable of analyzing a system defined in HLPNs automatically. We have applied our tool to analyze the safety properties of HLPN models of various problems from Model Checking Contest @ Petri Nets, the Mondex system, and the counter model. PIPE+Verifier is an open source tool and is available for sharing and continuous enhancements from worldwide research community.

Acknowledgments. This work was partially supported by NSF grants HRD-0833093 and IIP-1237818.

References

1. Cpn tools, `http://cpntools.org`
2. High-level Petri Nets - Concepts, Definitions and Graphical Notation (2000)
3. Armando, A., Mantovani, J., Platania, L.: Bounded model checking of software using smt solvers instead of sat solvers. Int. J. Softw. Tools Technol. Transf. 11(1), 69–83 (2009)
4. Armando, A., Ranise, S., Rusinowitch, M.: A rewriting approach to satisfiability procedures. Information and Computation 183(2), 140–164 (2003); 12th International Conference on Rewriting Techniques and Applications (RTA 2001)
5. Ball, T., Bounimova, E., Kumar, R., Levin, V.: Slam2: static driver verification with under 4. In: Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Austin, TX, pp. 35–42. FMCAD Inc. (2010)
6. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011)
7. Barrett, C., De Moura, L., Stump, A.: Design and results of the 1st satisfiability modulo theories competition (smt-comp.). Journal of Automated Reasoning 35, 2005 (2005)
8. Barrett, C., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library, SMT-LIB (2010), `http://www.SMT-LIB.org`

9. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
10. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT Solver. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 150–153. Springer, Heidelberg (2010)
11. Chiola, G., Franceschinis, G.: Colored gspn models and automatic symmetry detection. In: PNPM, pp. 50–60 (1989)
12. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT Solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013)
13. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. In: Formal Methods in System Design, p. 2001. Kluwer Academic Publishers (2001)
14. Clarke, E., Kroning, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
15. Clarke, E., Kroning, D., Ouaknine, J., Strichman, O.: Completeness and complexity of bounded model checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 85–96. Springer, Heidelberg (2004)
16. De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Commun. ACM 54(9), 69–77 (2011)
17. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
18. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM 17(11), 643–644 (1974)
19. Dutertre, B., De Moura, L.: The yices smt solver 2, 2 (2006), Tool paper, `http://yices.csl.sri.com/tool-paper.pdf`
20. Allen Emerson, E., Lei, C.-L.: Modalities for model checking: branching time logic strikes back. Sci. Comput. Program. 8(3), 275–306 (1987)
21. Fronc, Ł., Duret-Lutz, A.: LTL model checking with neco. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 451–454. Springer, Heidelberg (2013)
22. Holzmann, G.: Spin model checker, the: primer and reference manual, 1st edn. Addison-Wesley Professional (2003)
23. Hostettler, S., Marechal, A., Linard, A., Risoldi, M., Buchs, D.: High-level petri net model checking with alpina. Fundam. Inf. 113(3-4), 229–264 (2011)
24. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
25. Jensen, K., Kristensen, L.M., Wells, L.: Coloured petri nets and cpn tools for modelling and validation of concurrent systems. Int. J. Softw. Tools Technol. Transf. 9(3), 213–254 (2007)
26. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
27. Kordon, F., Linard, A., Becutti, M., Buchs, D., Fronc, L., Hulin-Hubard, F., Legond-Aubry, F., Lohmann, N., Marechal, A., Paviot-Adet, E., Pommereau, F., Rodrígues, C., Rohr, C., Thierry-Mieg, Y., Wimmel, H., Wolf, K.: Web report on the model checking contest @ petri net 2013 (June 2013), `http://mcc.lip6.fr`

28. Kordon, F., Linard, A., Beccuti, M., Buchs, D., Fronc, L., Hillah, L.-M., Hulin-Hubard, F., Legond-Aubry, F., Lohmann, N., Marechal, A., Paviot-Adet, E., Pommereau, F., Rodríguez, C., Rohr, C., Thierry-Mieg, Y., Wimmel, H., Wolf, K.: Model checking contest @ petri nets, report on the 2013 edition. CoRR, abs/1309.2485 (2013)
29. Kröning, D., Rümmer, P., Weissenbacher, G.: A proposal for a theory of finite sets, lists, and maps for the smt-lib standard. In: Informal proceedings, 7th International Workshop on Satisfiability Modulo Theories at CADE 22 (2009)
30. Liu, S., Zeng, R., He, X.: Pipe+ - a modeling tool for high level petri nets. In: SEKE, pp. 115–121 (2011)
31. Liu, S., Zeng, R., Sun, Z., He, X.: Samat - a tool for software architecture modeling and analysis. In: SEKE, pp. 352–358 (2012)
32. de Moura, L., Bjørner, N.: Satisfiability Modulo Theories: An Appetizer. In: Oliveira, M.V.M., Woodcock, J. (eds.) SBMF 2009. LNCS, vol. 5902, pp. 23–36. Springer, Heidelberg (2009)
33. Srivastava, S., Gulwani, S., Foster, J.S.: VS^3: SMT Solvers for Program Verification. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 702–708. Springer, Heidelberg (2009)
34. Stepney, S.: An Electronic Purse: Specification, Refinement, and Proof. Technical monograph. Oxford University Computing Laboratory, Programming Research Group (2000)
35. Stump, A., Barrett, C.W., Dill, D.L.: A decision procedure for an extensional theory of arrays. In: 16th IEEE Symposium on Logic in Computer Science, pp. 29–37. IEEE Computer Society (2001)
36. Ullman, J.D.: Elements of ML programming (ML97 ed.). Prentice-Hall, Inc., Upper Saddle River (1998)
37. Veanes, M., Bjørner, N., Gurevich, Y., Schulte, W.: Symbolic bounded model checking of abstract state machines. Int. J. Software and Informatics 3(2-3), 149–170 (2009)
38. Veanes, M., Bjørner, N.S., Raschke, A.: An SMT approach to bounded reachability analysis of model programs. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 53–68. Springer, Heidelberg (2008)
39. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engg. 10(2), 203–232 (2003)
40. Woodcock, J.: First steps in the verified software grand challenge. Computer 39(10), 57–64 (2006)
41. Woodcock, J., Stepney, S., Cooper, D., Clark, J.A., Jacob, J.: The certification of the mondex electronic purse to itsec level e6. Formal Asp. Comput. 20(1), 5–19 (2008)
42. Zeng, R., He, X.: Analyzing a formal specification of mondex using model checking. In: Cavalcanti, A., Deharbe, D., Gaudel, M.-C., Woodcock, J. (eds.) ICTAC 2010. LNCS, vol. 6255, pp. 214–229. Springer, Heidelberg (2010)
43. Zeng, R., Liu, J., He, X.: A formal specification of mondex using sam. In: IEEE International Symposium on Service-Oriented System Engineering, SOSE 2008, pp. 97–102 (December 2008)