A Simple Abstract Interpretation for Petri Net Queries

Karsten Wolf

Universität Rostock, Institut für Informatik
karsten.wolf@uni-rostock.de

Abstract. We propose a simple but effective method for the detection of duplicates in state predicates for Petri nets. We map sub-formulas to integer numbers using a mapping abstr. The assignment guarantees that $\text{abstr}(\phi_1) = \text{abstr}(\phi_2)$ implies equivalence of ϕ_1 and ϕ_2 while $\text{abstr}(\phi_1) = -\text{abstr}(\phi_2)$ implies equivalence of ϕ_1 and $\neg\phi_2$. The obtained knowledge can be used for reducing the size of a formula that is obtained by expensive constructions such as the transformation into disjunctive normal form. The method is implemented in our tool LoLA 2.0.

1 Introduction

Formulas in temporal logic [4], as used in model checking [1], are based on state predicates. For Petri nets, state properties refer to places (e.g. "the marking on place p is less than 2"), transitions (e.g. "transition t is fireable"), or the whole marking (e.g. "marking m is a deadlock"). They can be aggregated using Boolean operations and form the basis of temporal logics such as CTL or LTL.

Several tasks in model checking and other verification approaches require an involved rewriting process on state predicates. For example, one can use the state equation of Petri nets as a tool for verifying reachability [6]. This approach, however, works only for convex formulas (i.e. conjunctions of inequations and equations). For applying the method to a general formula, it needs to be rewritten into disjunctive normal form. Then every convex sub-formula can be checked separately using the state equation approach. Rewriting may lead to a replication of sub-formulas. We propose a simple but quite effective abstract interpretation that supports an easy detection of a large number of duplicates during state property transformation. This way, resulting state predicates become smaller and evaluating them during state space verification takes less time. We have implemented the method in our tool LoLA 2.0 [5, 7] available at `service-technology.org`.

2 Petri nets

We consider place/transition nets. A place/transition net N consists of finite and disjoint sets P (places) and T (transitions), an arc relation $F \subseteq (P \times T) \cup (T \times P)$, a weight function $W : F \to \mathbb{N} \setminus \{0\}$, and an initial marking m_0. A marking m is

a mapping $m : P \to \mathbb{N}$. Transition t is enabled in m iff, for all $p \in P$, $[p,t] \in F$ implies $m(p) \geq W([p,t])$.

We skip the definition of the behaviour of Petri nets since this is not relevant for this paper.

3 State Predicates

A state predicate ϕ is a logical formula that represents a property of markings. Applied to a marking, it yields true or false. State predicates are used for the specification of verification queries, typically as part of a formula in some temporal logic. Most verification tools for Petri nets support the specification of state predicates in one or the other syntax. Most competitions in the model checking contest (MCC) [3] involve state predicates. Our considerations are based on the following syntax and semantics.

Definition 1 (State predicate). *A state predicate can have any of the following shapes (k_i are integer numbers, p_i are places, t_i transitions, $\sim \in \{<, >, \leq, \geq, =, \neq\}$, and ϕ_i state predicates):*

- *TRUE, FALSE: true for all markings (no marking, respectively);*
- *$k_1 p_1 + \cdots + k_n p_n \sim k_0$: true for marking m iff $k_1 m(p_1) + \cdots + k_n m(p_n) \sim k_0$;*
- *FIREABLE(t_i): true for m iff t_i is enabled in m;*
- *DEADLOCK: true for m iff no transition is enabled in m;*
- *INITIAL: true for m iff $m = m_0$;*
- *$\phi_1 \wedge \cdots \wedge \phi_n$, $\phi_1 \vee \cdots \vee \phi_n$, $\neg\phi_1$, $\phi_1 \implies \phi_2$, $\phi_1 \iff \phi_2$: evaluated according to the usual interpretation of Boolean logic.*

State predicates can be represented as a tree, with the Boolean operators as inner nodes and the remaining predicates (called *atomic*) as leaves. We assume that a conjunction $\phi_1 \wedge \cdots \wedge \phi_n$ is represented as a single node with n children for the sub-formulas ϕ_i. The same applies to disjunctions.

4 Sources of duplicates

There are several transformations that involve replication of subformulas:

Rewriting Boolean operators. Even if all Boolean operators are permitted in specifications, internal representations are usually restricted to conjunction and disjunction. That is, $\phi_1 \iff \phi_2$ may be replaced with $(\phi_1 \wedge \phi_2) \vee (\neg\phi_1 \wedge \neg\phi_2)$ and causes a replication of sub-formulas ϕ_1 and ϕ_2. Similar rewriting rules apply to formulas in temporal logic, causing replication of involved state predicates.

Rewriting atomic predicates. Tools typically try to internally use as few as possible atomic state predicates. So FIREABLE(t) can be replaced by $\bigwedge_{p:[p,t]\in F} p \geq W([p,t])$. INITIAL can be replaced with $\bigwedge_{p\in P} p = m_0(p)$. DEADLOCK is equivalent to $\bigwedge_{t\in T} \neg$ FIREABLE(t). The overwhelming majority of Petri net models is ordinary (all arc weights are equal to one) and a considerable amount

of models is safe (at most one token on any place in all reachable markings). For example, the "known models" of the MCC in 2017 comprised of 65 ordinary nets versus 12 non-ordinary nets. 39 nets are safe, compared to 38 non-safe net models.

predicates like $p > 0$ and $p = 1$ may occur several times (e.g. once for every transition connected to p).

Frequently used predicates. Many specifications frequently use expressions like "place p is marked ($p > 0$)" or "p is unmarked ($p = 0$)". Hence, these predicates may occur more than once in a predicate.

Transformation into disjunctive normal form. Verification techniques that are based on linear algebra, for instance the exploitation of the state equation [6], are restricted to convex properties (i.e. conjunctions of inequations). Hence, FIREABILITY and other predicates must be unfolded and the resulting property must be transformed into a disjunction of conjunctions of inequations. Then, the individual conjunctions can be fed to the state equation separately. Generation of a disjunctive normal form involves the application of the law of distributivity: $(\phi_1 \vee \phi_2 \vee \phi_3) \wedge (\phi_4 \vee \phi_5)$ is to be rewritten to $(\phi_1 \wedge \phi_4) \vee (\phi_1 \wedge \phi_5) \vee (\phi_2 \wedge \phi_4) \vee (\phi_2 \wedge \phi_5) \vee (\phi_3 \wedge \phi_4) \vee (\phi_3 \wedge \phi_5)$.

Translation from other formalisms. Place/transition nets representing interesting systems, together with their verification queries, are often generated by translation from other specification languages. Translations are based on patterns and often induce duplicates of state predicates or sub-formulas. Consider, for instance, the translation from a place p of a Coloured Petri Net with domain $\{1, 2, 3\}$ into three places p_1, p_2, and p_3 of a place/transition net. Assuming that addition is not supported, formula $p =$ (three tokens on p) could be translated into

$$(p_1 = 0 \wedge p_2 = 0 \wedge p_3 = 3) \vee (p_1 = 0 \wedge p_2 = 1 \wedge p_3 = 2) \vee \cdots \vee (p_1 = 3 \wedge p_2 = 0 \wedge p_3 = 0)$$

This formula contains a lot of replications of elementary comparisons.

Consequently, a verification tool may expect the presence of duplicates in the state predicates it is given for verification. It generates additional duplicates in its internal transformations. Since place/transition nets are a formalism with low-level modelling primitives (transitions actually represent simple vector addition), automatically generated predicates may involve huge conjunctions and disjunctions. Generation of disjunctive normal form may easily explode. Detection of duplicates helps in reducing the size of resulting formulas thus speeding up subsequent use of the state predicates.

5 Abstract Interpretation

The idea of abstract interpretation [2] is to map certain entities into another (abstract) domain where some considerations can be verified more easily. Abstraction must be faithful (every result derived in the abstract domain must be correct in the original domain) but not necessarily precise (there may be results that hold in the original domain but cannot be derived in the abstract domain).

Loss of precision is the price to be paid for easier approaches in the abstract domain.

We propose to use the set $\mathbb{Z}$ of integer numbers as abstract domain. We aim at assigning a number $\text{abstr}(\phi)$ to every state predicate ϕ such that the following results may be reliably derived:

- if $\text{abstr}(\phi_1) = \text{abstr}(\phi_2)$ then ϕ_1 is semantically equivalent to ϕ_2;
- if $\text{abstr}(\phi_1) = -\text{abstr}(\phi_2)$ then ϕ_1 is semantically equivalent to $\neg\phi_2$;
- if $\text{abstr}(\phi_1) = 1$ then ϕ_1 is a tautology (true for every marking);
- if $\text{abstr}(\phi_1) = -1$ then ϕ_1 is a contradiction (false for every marking);

We do not attempt to meet the converse of any of these statements. That is, there may be semantically equivalent formulas with different values for their abstraction.

We explicitly store $\text{abstr}(\phi)$ for every state predicate ϕ and every of its subformulas. Hence, once computed, $\text{abstr}(\phi)$ can be derived from ϕ in constant time.

6 Abstraction of Atomic State Predicates

In this section, we discuss the assignment of abstract values to atomic state predicates (all but the Boolean combinations). The following assignment reflects the frequent occurrence of certain propositions. Without loss of generality, we assume that $P = \{p_1, \ldots, p_m\}$ and $T = \{t_1, \ldots, t_n\}$.

- $\text{abstr}(\text{TRUE}) := 1$;
- $\text{abstr}(\text{FALSE}) := -1$;
- $\text{abstr}(\text{DEADLOCK}) := 2$;
- $\text{abstr}(\text{INITIAL}) := 3$;
- for all $p_i \in P$, $\text{abstr}(p_i > 0) := 3 + i$ and $\text{abstr}(p_i = 0) := -3 - i$;
- for all $t_j \in T$, $\text{abstr}(\text{FIREABLE}(t_j)) := m + 3 + j$.

For the remaining atomic propositions, we assign consecutive numbers. To this end, we introduce a "function" new() that returns, whenever called, a fresh (previously unused) number greater than $3 + m + n$. That is, if any comparison, except the ones just mentioned, is used twice in a state predicate, the two incarnations get different values (and we do not detect their equality).

- $\text{abstr}(k_1 p_1 + \cdots + k_n p_n \sim k_0) := \text{new}()$;

For soundness, it is important that indeed every call to new() returns a distinct value. In a threaded context, this can be achieved without synchronisation: let every call to new() in thread k of an n-threaded program return only values that are congruent to k modulo n. Obviously, the proposed setting meets the requirements mentioned in Section 5.

7 Boolean combinations

For simplicity, we consider only conjunction, disjunction, and negation. We basically assume that equivalence and implication leave the scene early in the game using well-known rewrite rules. Negation can be handled quite easily. abstr($\neg\phi$) can be naturally set to $-\,\mathrm{abstr}(\phi)$, so the value can be inherited from the subformula.

Next we consider a conjunction with k sub-formulas. We proceed inductively, starting with a conjunction that has 0 sub-formulas. During this process, we combine the definition of the abstract value of the conjunction with the detection of duplicates among the subformulas. An empty conjunction is treated as TRUE, so it receives abstract value 1. Assume that we want to add ϕ_i to a given conjunction $C = \phi_1 \wedge \cdots \wedge \phi_{i-1}$. We distinguish the following cases:

- abstr(C) = 1: Then abstr($C \wedge \phi_i$) := abstr(ϕ_i), and $\phi_1, \ldots, \phi_{i-1}$ are removed from the formula;
- abstr(C) = -1: Then abstr($C \wedge \phi_i$) := -1, and ϕ_i is not added to the list of subformulas of C;
- abstr(ϕ_i) = 1: Then abstr($C \wedge \phi_i$) := abstr(C), and ϕ_i is not added to the list of subformulas of C;
- abstr(ϕ_i) = -1: Then abstr($C \wedge \phi_i$) := -1, and $\phi_1, \ldots, \phi_{i-1}$ are removed from the formula;
- abstr(C) = abstr(ϕ_i): Then abstr($C \wedge \phi_i$) := abstr(C), and ϕ_i is not added to the list of subformulas of C;
- abstr(C) = $-\,$abstr(ϕ_i): Then abstr($C \wedge \phi_i$) := -1;
- There is a $j < i$ with abstr(ϕ_j) = abstr(ϕ_i). Then abstr($C \wedge \phi_i$) := abstr(C), and ϕ_i is not added to the list of subformulas of C;
- There is a $j < i$ with abstr(ϕ_j) = $-\,$abstr(ϕ_i). Then abstr($C \wedge \phi_i$) := -1;
- Otherwise, abstr($C \wedge \phi_i$) := new().

The assignment reflects simple laws of Boolean logic: $\phi \wedge \phi$ is ϕ and $\phi \wedge \neg\phi$ is false. Hence, the assignment reflects the specification in Section 5. The approach relies on the assumption that formulas are constructed only by adding sub-formulas. It does not work for a process that involves the removal of a sub-formula.

For a disjunction, the assignment can be naturally derived from the considerations for conjunction and negation: $\phi_1 \vee \phi_2$ is equivalent to $\neg(\neg\phi_1 \wedge \neg\phi_2)$. In particular, the empty disjunction should get value -1.

Using the above definition, the abstract value may depend on the order in which subformulas appear in a conjunction or disjunction. Assuming abstr(ϕ_1) = 10, abstr(ϕ_2) = -10, and abstr(ϕ_3) = 25, then abstr($\phi_1 \wedge \phi_2 \wedge \phi_3$) = -1 while abstr($\phi_1 \wedge \phi_3 \wedge \phi_2$) would be the result of another call to new(), e.g. 1001. Both values are correct but naturally -1 is the preferable value. Hence, a clever implementation would always assign -1 when values k and $-k$ appear as abstract values in the list of subformulas of a conjunction. Disjunctions can be handled similarly.

The described process can be used for updating abstract values while composing a formula from its constituents. This is the case for the original construction

of the predicate, but also for the construction of a disjunctive normal form. There are a few constructions where we assign the abstract value differently. When we copy a predicate (for instance when rewriting an equivalence), the copy gets the same abstract value as the original predicate. When we transform a FIREABILITY, DEADLOCK, or INITIAL predicate into a Boolean combination of inequations, the resulting predicate inherits the abstract value of the original formula as well. When we turn a predicate into its negation (e.g. when applying de Morgan's rule for eliminating negation symbols), the resulting predicate gets the inverse of the original value.

8 Examples

Consider the formula $p = 1 \iff q = 0$ where p and q are assumed to be places. Internally, we support only $\leq$, so we would rewrite the formula to $(p \leq 1 \wedge 1 \leq p) \iff q \leq 0$ (assume that we detect $q \geq 0$ to be a tautology since q cannot get a negative number of tokens). Further, we normalise the formula such that only $\wedge$ and $\vee$ is used. We apply the rewrite rule that replaces $A \iff B$ with $(A \vee \neg B) \wedge (\neg A \vee B)$. We obtain:

$$((p \leq 1 \wedge 1 \leq p) \vee 1 \leq q) \wedge (p \leq 0 \vee 2 \leq p \vee q \leq 0)$$

This is the formula that we normally use for state space verification. Since the rewrite rule introduced *copies* of A and B, the two copies (and their subformulas) get the same value which is than inversed through negation in one of the two copies. This way, we preserve some knowledge about the origin of sub-formulas.

If, subsequently, we want to include the state equation (i.e. linear programming) approach, we transform the formula into a disjunctive normal form since linear program can handle only conjunctions of inequations. We proceed by first transforming the left part, resulting in

$$(p \leq 1 \vee 1 \leq q) \wedge (1 \leq p \vee 1 \leq q) \wedge (p \leq 0 \vee 2 \leq p \vee q \leq 0)$$

Then we apply the law of distributivity to the whole formula. Doing that in the brute-force way, would result in in a disjunction of 12 subformulas:

$$\begin{array}{l}
(p \leq 1 \wedge 1 \leq p \wedge p \leq 0) \vee \\
(p \leq 1 \wedge 1 \leq p \wedge 2 \leq p) \vee \\
(p \leq 1 \wedge 1 \leq p \wedge q \leq 0) \vee \\
(p \leq 1 \wedge 1 \leq q \wedge p \leq 0) \vee \\
(p \leq 1 \wedge 1 \leq q \wedge 2 \leq p) \vee \\
(p \leq 1 \wedge 1 \leq q \wedge q \leq 0) \vee \\
(1 \leq q \wedge 1 \leq p \wedge p \leq 0) \vee \\
(1 \leq q \wedge 1 \leq p \wedge 2 \leq p) \vee \\
(1 \leq q \wedge 1 \leq p \wedge q \leq 0) \vee \\
(1 \leq q \wedge 1 \leq q \wedge p \leq 0) \vee \\
(1 \leq q \wedge 1 \leq q \wedge 2 \leq p) \vee \\
(1 \leq q \wedge 1 \leq q \wedge q \leq 0)
\end{array}$$

Our abstraction, however, recognises that $p \leq 0$ is the negation of $1 \leq p$, $p \leq 1$ is the negation of $2 \leq p$, and $q \leq 0$ is the negation of $1 \leq q$ since the abstraction of one the the formulas is -1 times the abstraction of the other one. The detection does not depend on the simplicity of these formulas since the abstraction takes care of copying and negation in the process of transforming the original formula. As soon as a formula and its negation appear in a conjunction, the conjunction is replaced by false and thus disregarded in the resulting formula. This way, first, second, 5th, 6th, 7th, 9th, and 12th subformulas Furthermore it recognises the duplicate appearance of $1 \leq q$ in the 10th and 11th sub-formulas, so we would produce the normal form

$$\begin{array}{l}(p \leq 1 \wedge 1 \leq p \wedge q \leq 0) \vee \\ (p \leq 1 \wedge 1 \leq q \wedge p \leq 0) \vee \\ (1 \leq q \wedge 1 \leq p \wedge 2 \leq p) \vee \\ (1 \leq q \wedge p \leq 0) \vee \\ (1 \leq q \wedge 2 \leq p)\end{array}$$

Our approach reduced a formula with 36 literals to a formula with 13 literals. Instead of 12 conjunctions, only 5 conjunctions need to be shipped to the linear programming tool. One might argue that formula would not have exploded in the first place if the equivalence $A \iff B$ would have been rewritten by $(A \wedge B) \vee (\neg A \wedge \neg B)$. Using that rule, however, the negated formula would explode instead, so the choice of the rule for rewriting equivalence does not solve the problem of duplication.

9 Experience and Conclusion

We implemented the abstract interpretation in our tool LoLA 2 [5, 7]. Our main focus was to reduce the size of disjunctive normal form generation which is a pre-requisite for applying the state equation based method of [6]. The abstract interpretation yields a sufficient condition that sub-formulas can be merged. We jointly apply a hash-based technique that yields a sufficient condition for sub-formulas to be at least syntactically different. Whenever none of the two criteria applies, we investigate the sub-formulas recursively to look for additional duplicates. The abstract interpretation costs virtually no time, so we are convinced that it establishes a nice tool for handling state predicates in verification. Using special codes for frequently occurring atomic predicates, the method is in fact Petri net specific.

References

1. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking.* MIT Press, 2001.
2. Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

3. F. Kordon et al. Homepage of the Model Checking Contest. http://mcc.lip6.fr/, June 2017.
4. Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - specification.* Springer, 1992.
5. K. Schmidt. Lola: A low level analyser. In *ICATPN, LNCS 1825*, pages 465–474, 2000.
6. H. Wimmel and K. Wolf. Applying CEGAR to the Petri net state equation. *Logical Methods in Computer Science*, 8(3), 2012.
7. K. Wolf. Petri net model checking with lola 2. In *Accepted for: Petri Nets 2018*, pages 465–474, 2000.