Shortest Violation Traces in Model Checking Based on Petri Net Unfoldings and SAT[⋆]

Victor Khomenko

School of Computing Science, University of Newcastle
Newcastle upon Tyne NE1 7RU, U.K.
e-mail: `Victor.Khomenko@ncl.ac.uk`

Abstract. Model checking based on the causal partial order semantics of Petri nets is an approach widely applied to cope with the state space explosion problem. One of the possibilities for the verification process is to build a finite and complete prefix and use it for constructing a Boolean formula such that any satisfying assignment to its variables yields a trace violating the property being checked. (And if there are no satisfying assignments then the property holds.)
In this paper a method for computing the *shortest* violation traces (which can greatly facilitate debugging) is proposed. Experimental results demonstrate that it can achieve significant reductions in the size of the Boolean formula as well as in the time required to compute a shortest violation trace, when compared with a naïve approach.
Keywords: Shortest trace, model checking, Petri net unfolding, SAT, Boolean circuit.

1 Introduction and basic notions

A distinctive characteristic of reactive concurrent systems is that their sets of local states have descriptions which are both short and manageable, and the complexity of their behaviour comes from highly complicated interactions with the external environment rather than from complicated data structures and manipulations thereon. One way of coping with this complexity problem is to use formal methods and, especially, computer aided verification tools implementing model checking — a technique in which the verification of a system is carried out using a finite representation of its state space.

The main drawback of model checking is that it suffers from the *state space explosion* problem. That is, even a relatively small system specification can (and often does) yield a very large state space. To cope with this, several techniques have been developed, which usually aim either at a compact representation of the full state space of the system, or at the generation of its reduced (though sufficient for a given verification task) state space. Among them, a prominent technique is McMillan's (finite prefixes of) Petri Net unfoldings (see, e.g., [5, 7]). They rely on the partial order view of concurrent computation, and represent system states implicitly, using an acyclic net, called a *prefix*.

Most of 'interesting' problems for safe Petri nets are $\mathcal{PSPACE}$-complete [2], but the same problems for prefixes are often in $\mathcal{NP}$ or even $\mathcal{P}$. Though the size

[⋆] The full version of this paper [6] is available on-line.

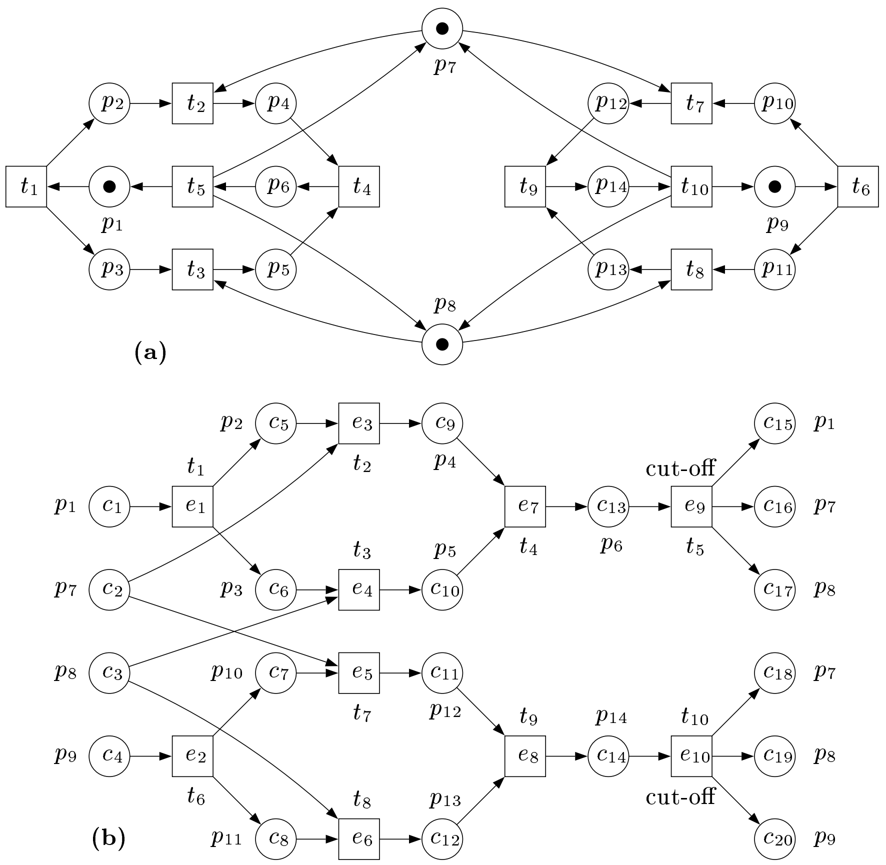
of a finite and complete unfolding prefix can be exponential in the size of the original Petri net, in practice it is often relatively small.

A model checking problem formulated for a prefix can usually be translated into some canonical problem, e.g., Boolean satisfiability (SAT). Then an off-the-shelf SAT solver can be used for efficiently solving it. Such a combination 'unfolder & solver' turns out to be quite powerful in practice.

Petri nets A *net* is a triple $N \overset{\text{df}}{=} (P, T, F)$ such that P and T are disjoint sets of respectively *places* and *transitions*, and $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation*. A *marking* of N is a multiset M of places, i.e., $M : P \to \mathbb{N} \overset{\text{df}}{=} \{0, 1, 2, \ldots\}$. The standard rules about drawing nets are adopted in this paper, viz. places are represented as circles, transitions as boxes, the flow relation by arcs, and the marking is shown by placing tokens within circles. As usual, ${}^\bullet z \overset{\text{df}}{=} \{y \mid (y, z) \in F\}$ and $z^\bullet \overset{\text{df}}{=} \{y \mid (z, y) \in F\}$ denote the *pre-* and *postset* of $z \in P \cup T$, and ${}^\bullet Z \overset{\text{df}}{=} \bigcup_{z \in Z} {}^\bullet z$ and $Z^\bullet \overset{\text{df}}{=} \bigcup_{z \in Z} z^\bullet$, for all $Z \subseteq P \cup T$. In this paper, the presets of transitions are restricted to be non-empty, i.e., ${}^\bullet t \neq \emptyset$ for every $t \in T$. A *net system* is a pair $\Upsilon \overset{\text{df}}{=} (N, M_0)$ comprising a finite net N and an *initial* marking M_0. It is assumed that the reader is familiar with the standard notions of the Petri nets theory, such as the *enabledness* and *firing* of a transition, marking *reachability* and *deadlock*.

Unfolding prefix A *finite and complete unfolding prefix* π of a Petri net Υ is a finite acyclic net which implicitly represents all the reachable states of Υ together with transitions enabled at those states. Intuitively, it can be obtained through *unfolding* Υ, by successive firings of transition, under the following assumptions: (a) for each new firing a fresh transition (called an *event*) is generated; (b) for each newly produced token a fresh place (called a *condition*) is generated. The unfolding is infinite whenever Υ has an infinite run; however, if Υ has finitely many reachable states then the unfolding eventually starts to repeat itself and can be truncated (by identifying a set of *cut-off* events) without loss of information, yielding a finite and complete prefix. The sets of conditions, events and cut-off events of the prefix are denoted by B, E and E_{cut}, respectively. (Note that $E_{cut} \subseteq E$).

Efficient algorithms exist for building such prefixes [5], which ensure that the number of non-cut-off events $|E \setminus E_{cut}|$ in a complete prefix can never exceed the number of reachable states of Υ. Moreover, complete prefixes are often exponentially smaller than the corresponding state graphs, especially for highly concurrent Petri nets, because they represent concurrency directly rather than by multidimensional 'diamonds' as it is done in state graphs. For example, if the original Petri net consists of 100 transitions which can fire once in parallel, the state graph will be a 100-dimensional hypercube with 2^{100} vertices, whereas the complete prefix will coincide with the net itself. Another example, viz. a Petri net modelling two dining philosophers, and a finite and complete prefix of its unfolding, are shown in Fig. 1. One can observe that if this example is scaled up, the size of the prefix is linear in the number of dining philosophers, even though the number of reachable states grows exponentially.

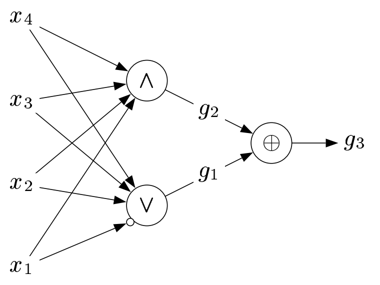


(a)
p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9 p_{10} p_{11} p_{12} p_{13} p_{14}
t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10}
(b)
c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8 c_9 c_{10} c_{11} c_{12} c_{13} c_{14} c_{15} c_{16} c_{17} c_{18} c_{19} c_{20}
e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9 e_{10}
cut-off
cut-off

Fig. 1. A Petri net modelling two dining philosophers **(a)** and a finite and complete prefix of its unfolding **(b)**.

Since π is acyclic, the transitive closure of its flow relation is a partial order $<$ on $B \cup E$, called the *causality relation*. (The reflexive order corresponding to $<$ will be denoted by $\leq$.) Intuitively, all the events which are smaller than an event $e \in E$ w.r.t. $<$ must precede e in any valid execution containing e. Two nodes $x, y \in B \cup E$ are in *conflict*, denoted $x\#y$, if there are distinct events $e, f \in E$ such that ${}^\bullet e \cap {}^\bullet f \neq \emptyset$ and $e \leq x$ and $f \leq y$. Intuitively, no valid execution can contain two events in conflict. Two nodes $x, y \in B \cup E$ are *concurrent*, denoted $x \; co \; y$, if neither $y\#y'$ nor $y \leq y'$ nor $y' \leq y$. Intuitively, two concurrent events can be enabled simultaneously, and executed in any order, or even concurrently. For example, in the prefix shown in Fig. 1(b) the following relationships hold: $e_1 < e_7$, $e_7\#e_8$ (due to the choices at c_2 and c_3) and $e_3 \; co \; e_4$.

The reachable markings of Υ can be represented using *configurations* of π. A *configuration* is a set of events $C \subseteq E \setminus E_{cut}$ such that for all $e, f \in C$, $\neg(e\#f)$ and, for every $e \in C$, $f < e$ implies $f \in C$. For example, in the net shown in Fig. 1(b), $\{e_1, e_3, e_4\}$ is a configuration, whereas $\{e_1, e_2, e_3, e_5\}$ and $\{e_1, e_3, e_7\}$



$$\begin{array}{l}
[g_1 \leftrightarrow (\neg x_1 \vee x_2 \vee x_3 \vee x_4)] \wedge \\
[g_2 \leftrightarrow (x_1 \wedge x_2 \wedge x_3 \wedge x_4)] \wedge \\
[g_3 \leftrightarrow (g_1 \oplus g_2)] \\
\qquad\qquad \equiv \\
[(\neg g_1 \vee \neg x_1 \vee x_2 \vee x_3 \vee x_4) \wedge \\
(g_1 \vee x_1) \wedge (g_1 \vee \neg x_2) \wedge \\
(g_1 \vee \neg x_3) \wedge (g_1 \vee \neg x_4)] \wedge \\
[(g_2 \vee \neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge \\
(\neg g_2 \vee x_1) \wedge (\neg g_2 \vee x_2) \wedge \\
(\neg g_2 \vee x_3) \wedge (\neg g_2 \vee x_4)] \wedge \\
[(\neg g_1 \vee \neg g_2 \vee \neg g_3) \wedge (\neg g_1 \vee g_2 \vee g_3) \wedge \\
(g_1 \vee \neg g_2 \vee g_3) \wedge (g_1 \vee g_2 \vee \neg g_3)]
\end{array}$$

Fig. 2. Conversion of a Boolean circuit into a Boolean expression in the CNF.

are not (the former includes events in conflict, $e_3 \# e_5$, while the latter does not include $e_4 < e_7$). Intuitively, a configuration is a partial-order execution, i.e., an execution where the order of firing of some of its events (viz. concurrent ones) is not important; e.g., the configuration $\{e_1, e_3, e_4, e_7\}$ corresponds to two totally ordered executions: $e_1 e_3 e_4 e_7$ and $e_1 e_4 e_3 e_7$. Since a configuration can correspond to multiple executions, it is often much more efficient in model checking to explore configurations rather than executions.

After starting π from the implicit initial marking (whereby one puts a single token in each condition which does not have an incoming arc) and executing all the events in C, one reaches the marking denoted by $\mathit{Cut}(C)$. $\mathit{Mark}(C)$ denotes the corresponding marking of Υ, reached by firing a transition sequence corresponding to the events in C. It is remarkable that each reachable marking of Υ is $\mathit{Mark}(C)$ for some configuration C of π, and, conversely, each configuration C of π generates a reachable marking $\mathit{Mark}(C)$. Thus various behavioural properties of Υ can be re-stated as the corresponding properties of π, and then checked, often much more efficiently.

Boolean satisfiability The *Boolean satisfiability problem (SAT)* consists in finding a *satisfying assignment*, i.e., a mapping $A : \mathit{Var}_\varphi \to \{0, 1\}$ defined on the set of variables Var_φ occurring in a given Boolean expression φ such that φ evaluates to 1. This expression is often assumed to be given in the *conjunctive normal form (CNF)* $\varphi = \bigwedge_{i=1}^{n} \bigvee_{l \in L_i} l$, i.e., it is represented as a conjunction of *clauses*, which are disjunctions of *literals*, each literal l being either a variable or the negation of a variable. It is assumed that no two literals in the same clause correspond to the same variable.

In order to solve a Boolean satisfiability problem, SAT solvers perform exhaustive search assigning the values 0 or 1 to the variables, using heuristics to reduce the search space [10]. Some of the leading SAT solvers, e.g., zChaff [8], can be used in the *incremental mode,* i.e., after solving a particular SAT instance the user can slightly change it (e.g., by adding and/or removing a small number of clauses) and execute the solver again. This is often much more efficient than solving these related instances as independent problems, because on the subsequent runs the solver can use some of the useful information (e.g., learnt clauses [10]) collected so far.

Boolean circuits A *Boolean circuit* (see, e.g., [9]) computes a multiple-output Boolean function of Boolean *input variables* $x_1, \ldots, x_n$. It consists of a finite number k of *gates* $G_1, \ldots, G_k$. Each gate G_i is labelled by a Boolean function f_i chosen from some fixed set of Boolean functions $\mathcal{F}$. (In this paper, $\mathcal{F}$ comprises all the unary and binary Boolean functions and conjunctions and disjunctions of arbitrary arity with arbitrary input inversions.) A Boolean circuit can be represented by an acyclic directed graph, where the input variables and the constants 0 and 1 are its sources, and the vertex representing the gate G_i has $arity(f_i)$ numbered incoming edges from its predecessors in the graph. (If f_i is commutative, the numbering of edges does not have to be specified.) In pictures, each gate is represented as a circle with the function shown within it, and input inversions are shown as 'bubbles'. Note that $\mathcal{F}$ is closed w.r.t. input inversions, and so they can be incorporated into the corresponding gate function.

The Boolean function f_v computed at a vertex v of this acyclic graph is defined inductively as follows. If v is an input variable x_j then $f_v(x_1, \ldots, x_n) \overset{\text{df}}{=} x_j$, and if it is a constant $c \in \{0, 1\}$ then $f_v(x_1, \ldots, x_n) \overset{\text{df}}{=} c$. Otherwise, the vertex is some gate G_i, and $f_v(x_1, \ldots, x_n) \overset{\text{df}}{=} f_i(p_1, \ldots, p_{arity(f_i)})$, where $p_1, \ldots, p_{arity(f_i)}$ are the functions computed at the predecessors of this vertex in the graph. The *output vector* $(v_1, \ldots, v_m)$, where v_i is some vertex of the graph, describes what the circuit computes, viz. the multiple-output Boolean function $(f_{v_1}, \ldots, f_{v_m})$. In particular, any Boolean formula over the signature $\mathcal{F}$ can be represented as a circuit.

It turns out that a Boolean circuit can be efficiently encoded by a Boolean expression φ in the CNF depending on the variables Var_φ corresponding to the vertices of the graph representing the circuit (except 0 and 1) such that for any assignment $A : Var_\varphi \to \{0, 1\}$, A is a satisfying assignment of φ iff for every $v \in Var_\varphi$, $f_v(A(x_1), \ldots, A(x_n)) = A(v)$ (where the variables are denoted by the same symbol as the corresponding vertices of the graph) and $A(0) \overset{\text{df}}{=} 0$ and $A(1) \overset{\text{df}}{=} 1$.

The expression φ is constructed as follows. For each gate G_i, a new Boolean variable g_i representing its output is created, a Boolean equation relating g_i to the inputs of G_i is written down, and these equations are converted into the CNF. This process is illustrated in Fig. 2. Note that for a gate labelled with a Boolean function of bounded arity, the size of the corresponding equation (and its CNF) is bounded by a constant; moreover, for a gate labelled with a multiple-input conjunction or disjunction, the size of the equation (and its CNF) is linear in the number of gate inputs. Thus the size of the resulting Boolean expression in the CNF is linear in the size of the circuit.

Model checking based on Petri net unfoldings This paper concentrates on the following approach to model checking. First, a finite and complete prefix of the Petri net unfolding is built, and it is then used for constructing a Boolean formula encoding the model checking problem at hand. (It is assumed that the property being checked is the unreachability of some 'bad' states, e.g., deadlocks.) This formula is unsatisfiable iff the property holds, and such that any satisfying assignment to its variables yields a trace violating the property being checked.

Typically such a formula would have for each non-cut-off event e of the prefix a variable conf_e (the formula might also contain other variables), and for every satisfying assignment A, the set of events $C \overset{\mathrm{df}}{=} \{e \mid \mathsf{conf}_e = 1\}$ is a configuration such that $\mathit{Mark}(C)$ violates the property being checked. The formula often has the form $\mathcal{CONF} \wedge \mathcal{VIOL}$. The role of the *configuration constraint, $\mathcal{CONF}$*, is to ensure that C is a configuration of the prefix (not just an arbitrary set of events). $\mathcal{CONF}$ can be defined as the conjunction of the formulae

$$\bigwedge_{e \in E \setminus E_{cut}} \bigwedge_{f \in {}^\bullet({}^\bullet e)} (\mathsf{conf}_e \to \mathsf{conf}_f) \quad \text{and} \quad \bigwedge_{e \in E \setminus E_{cut}} \bigwedge_{f \in (({}^\bullet e)^\bullet \setminus \{e\}) \setminus E_{cut}} \neg(\mathsf{conf}_e \wedge \mathsf{conf}_f) \ .$$

The former formula ensures that if $e \in C$ then its immediate predecessors are also in C, i.e., C is downward closed w.r.t. $<$. The latter one ensures that C contains no conflicts. $\mathcal{CONF}$ can be transformed into the CNF by applying the rules $x \to y \equiv \neg x \vee y$ and $\neg(x \wedge y) \equiv \neg x \vee \neg y$. For example, the configuration constraint for the prefix shown in Fig. 1(b) is

$$\begin{aligned}&(\mathsf{conf}_{e_3} \to \mathsf{conf}_{e_1}) \wedge (\mathsf{conf}_{e_4} \to \mathsf{conf}_{e_1}) \wedge (\mathsf{conf}_{e_5} \to \mathsf{conf}_{e_2}) \wedge \\ &\quad (\mathsf{conf}_{e_6} \to \mathsf{conf}_{e_2}) \wedge (\mathsf{conf}_{e_7} \to \mathsf{conf}_{e_3}) \wedge (\mathsf{conf}_{e_7} \to \mathsf{conf}_{e_4}) \wedge \\ &\quad (\mathsf{conf}_{e_8} \to \mathsf{conf}_{e_5}) \wedge (\mathsf{conf}_{e_8} \to \mathsf{conf}_{e_6}) \wedge \neg(\mathsf{conf}_{e_3} \wedge \mathsf{conf}_{e_5}) \wedge \neg(\mathsf{conf}_{e_4} \wedge \mathsf{conf}_{e_6}) \ .\end{aligned}$$

The role of the *violation constraint, $\mathcal{VIOL}$*, is to express the property violation condition for a configuration C, so that if a configuration C satisfying this constraint is found then the property does not hold, and any ordering of events in C consistent with $<$ is a violation trace. For example, for deadlock checking $\mathcal{VIOL}$ can be defined as

$$\bigwedge_{e \in E} \Big(\bigvee_{f \in {}^\bullet({}^\bullet e)} \neg \mathsf{conf}_f \vee \bigvee_{f \in ({}^\bullet e)^\bullet \setminus E_{cut}} \mathsf{conf}_f \Big) \ .$$

This formula requires for each event e (including cut-off events) that some of the direct causal predecessors of e has not fired or some of the non-cut-off events (including e unless it is cut-off) consuming tokens from ${}^\bullet e$ has fired, and thus e is not enabled. This formula is already in the CNF. For example, the violation constraint for the deadlock checking problem formulated for the prefix shown in Fig. 1(b) is

$$\begin{aligned}&\mathsf{conf}_{e_1} \wedge \mathsf{conf}_{e_2} \wedge (\neg \mathsf{conf}_{e_1} \vee \mathsf{conf}_{e_3}) \wedge (\neg \mathsf{conf}_{e_1} \vee \mathsf{conf}_{e_4}) \wedge \\ &\quad (\neg \mathsf{conf}_{e_2} \vee \mathsf{conf}_{e_5}) \wedge (\neg \mathsf{conf}_{e_2} \vee \mathsf{conf}_{e_6}) \wedge (\neg \mathsf{conf}_{e_3} \vee \neg \mathsf{conf}_{e_4} \vee \mathsf{conf}_{e_7}) \wedge \\ &\quad (\neg \mathsf{conf}_{e_5} \vee \neg \mathsf{conf}_{e_6} \vee \mathsf{conf}_{e_8}) \wedge \neg \mathsf{conf}_{e_7} \wedge \neg \mathsf{conf}_{e_8} \ .\end{aligned}$$

Shortest violation traces Note that in general the computed violation trace can be quite long, which might make it difficult to locate the error, as the designer has to inspect this trace in order to find and eliminate the source of the problem. (And parts of such long traces often describe incidental system activity which is unrelated to the problem.) Thus computing shortest possible violation traces can greatly facilitate the debugging process.

A quite obvious algorithm for computing the shortest violation trace is shown in Fig. 3, where $\mathit{SAT\_Assignment}(\varphi)$ is a function computing a satisfying assignment for a Boolean formula φ and returning UNSAT in case φ is unsatisfiable (it is usually implemented by a call to some off-the-shelf SAT solver,

```
input : $\varphi$ — a Boolean formula
output : $T$ — the shortest violation trace or UNSAT

$A \leftarrow SAT\_Assignment(\varphi)$
if $A =$ UNSAT
    then
        $T \leftarrow$ UNSAT
        stop
$T \leftarrow Extract\_Trace(A)$
$r \leftarrow |T|$
$l \leftarrow 0$
while $l < r$ do
    $t \leftarrow \lceil (l + r)/2 \rceil$
    $A \leftarrow SAT\_Assignment(\varphi \wedge Threshold_t)$
    if $A =$ UNSAT
    then
        $l = t + 1$
    else
        $T \leftarrow Extract\_Trace(A)$
        $r \leftarrow |T|$
```

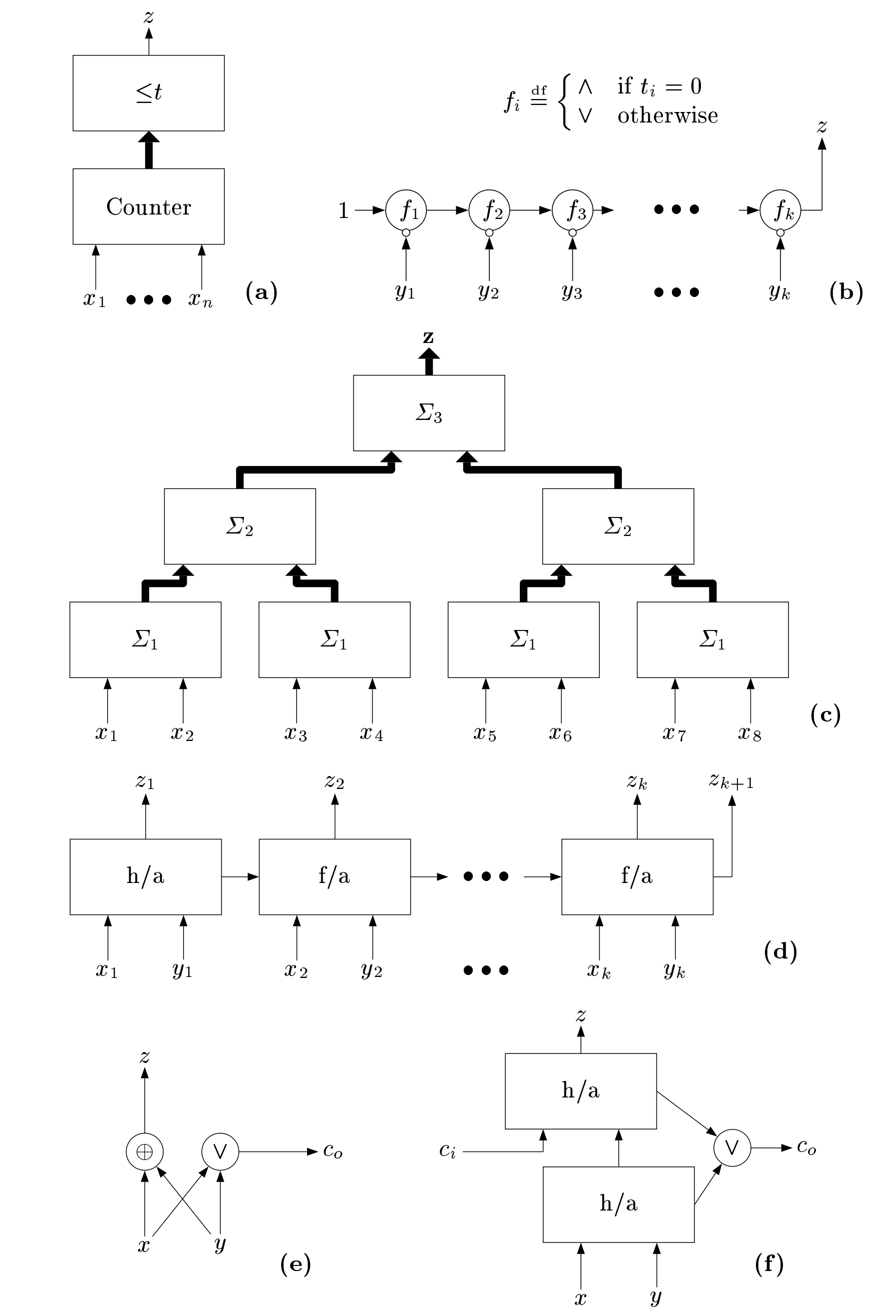
Fig. 3. An algorithm for computing shortest violation traces.

e.g., zChaff [8]), *Extract_Trace*(A) is a function extracting the violation trace from a satisfying Boolean assignment A, and *Threshold*$_t$ is the *threshold constraint* $|\{e \mid \mathsf{conf}_e = 1\}| \leq t$. This algorithm uses a binary search to compute the length of the shortest trace still exhibiting the violation. If the property holds (i.e., if φ is unsatisfiable) then this algorithm does not have any additional overhead compared with the original model checking algorithm, but in the case of errors the SAT solver is called several times with larger formulae, and so the overhead might be quite significant. This situation is somewhat alleviated by the fact that SAT instances are very similar to each other (in fact, even the formulae of the form *Threshold*$_t$, described in detail further in this paper, change very little when t changes) and thus can be efficiently solved in the incremental mode. Moreover, the user always can terminate the execution of the algorithm and get the shortest violation trace computed so far.

What still needs describing is the construction of the formula *Threshold*$_t$ for a given t. It turns out that one can exploit some problem-specific optimisations in order to significantly reduce the size of this formula as well as the computation effort required for solving the corresponding SAT instances. This is the main topic of this paper.

2 Basic translation of a threshold constraint

Threshold$_t$ can be expressed as a *pseudo-Boolean* constraint $\sum_{e \in E \setminus E_{cut}} \mathsf{conf}_e \leq t$, where arithmetical operations are used instead of logical ones. The other constraints can also be converted into a similar form, and the problem can be solved by a 0–1 integer linear programming solver. However, SAT solvers tend to be



z
$\leq t$
Counter
x_1 ●●● x_n
(a)
$f_i \overset{\mathrm{df}}{=} \begin{cases} \wedge & \text{if } t_i = 0 \\ \vee & \text{otherwise} \end{cases}$
z
1
f_1 f_2 f_3 ●●● f_k
y_1 y_2 y_3 ●●● y_k
(b)
z
Σ_3
Σ_2 Σ_2
Σ_1 Σ_1 Σ_1 Σ_1
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
(c)
z_1 z_2 z_k z_{k+1}
h/a f/a ●●● f/a
x_1 y_1 x_2 y_2 ●●● x_k y_k
(d)
z
$\oplus$ $\vee$ c_o
x y
(e)
z
h/a
c_i
$\vee$ c_o
h/a
x y
(f)

Fig. 4. Implementations of a threshold constraint **(a)**; a comparator **(b)**, where the inputs $y_1, \ldots, y_k$ are interpreted as the binary representation of a non-negative integer (least significant digit first) and $t_1, \ldots, t_k$ is the binary representation of t; a counter as a balanced tree of adders **(c)**; a k-bit adder Σ_k comprising a half-adder cell and $k-1$ full-adder cells **(d)**; and half-adder and full-adder cells **(e,f)**.

more efficient in practice, and so in many cases it would be advantageous to express $\mathit{Threshold}_t$ as a purely Boolean constraint.

A possible implementation of $\mathit{Threshold}_t$ as a Boolean circuit is shown in Fig. 4(a). It consists of two parts: the counter and the comparator. The counter circuit has n inputs and $\lceil \log_2 n \rceil + 1$ outputs, and its purpose is to count the number of ones among its inputs and return the result as a binary number. The purpose of the comparator is to compare this number with a given constant t.

Note that the counter circuit does not depend on t and so the corresponding part of the formula does not have to be changed between the calls to the SAT solver in the algorithm shown in Fig. 3. A possible implementation of the comparator is shown in Fig. 4(b). Note that it does depend on t, and so the corresponding part of the formula has to be amended from call to call. However, the size of the comparator is just $O(\log n)$. Thus this implementation of the threshold constraint is beneficial if the SAT solver is used in the incremental mode. The rest of this section is devoted to the counter circuit.

Fig. 4(c) illustrates an implementation of the counter as a tree of adders, where each adder is built of half-adder and full-adder cells, as shown in Fig. 4(d). A half-adder cell adds up two one-bit numbers, producing a one-bit result and a carry bit. A full-adder cell adds up two one-bit numbers and a carry from the previous cell of the adder, producing a one-bit result and a carry bit. Fig. 4(e,f) shows possible implementations of these cells.

The described circuit can be converted to a linear-size formula in the CNF, as described in Section 1. However, somewhat shorter formulae can be obtained using Boolean minimisation when translating half-adder and full-adder cells. It yields the formulae

$$(\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (x \vee y \vee \neg z) \wedge (y \vee \neg c_o) \wedge (\neg x \vee c_o \vee z) \wedge (\neg c_o \vee \neg z)$$

with 2 new variables, 6 clauses and 16 literals for a half-adder cell, and

$$(c_i \vee \neg x \vee y \vee z) \wedge (c_i \vee x \vee \neg y \vee z) \wedge (\neg c_i \vee \neg x \vee y \vee \neg z) \wedge (\neg c_i \vee x \vee \neg y \vee \neg z) \wedge (\neg c_i \vee c_o \vee z) \wedge$$
$$(c_i \vee \neg c_o \vee \neg z) \wedge (\neg x \vee \neg y \vee c_o) \wedge (x \vee y \vee \neg c_o) \wedge (\neg c_i \vee \neg x \vee \neg y \vee z) \wedge (c_i \vee x \vee y \vee \neg z)$$

with 2 new variables, 10 clauses and 36 literals for a full-adder cell.

It is shown in [6] that if n is a power of 2 then the resulting CNF formula for the counter contains $4n - 2\log_2 n - 4$ auxiliary variables (corresponding to gate outputs), $16n - 10\log_2 n - 16$ clauses and $52n - 36\log_2 n - 52$ literals, i.e., even though the size of the formula is linear in the number of the circuit's inputs, the multiplicative constants hidden in this $O(n)$ translation are quite large. Next section tries to remedy this situation by exploiting the structure of the prefix to improve the described translation.

3 Exploiting the structure of the prefix

The content of this section is the main contribution of this paper. It turns out that the structure of the prefix can be exploited to reduce the size of the counter circuit. Below, two heuristics are described, one utilising the conflicts between the events in the prefix, and the other making use of the causality relation.

Exploiting the conflicts One can observe that if $E' \subseteq E \setminus E_{cut}$ is a set of events which are in conflict with each other (i.e., E' is a clique in the graph corresponding to the relation #) then no two events from E' can belong to the same configuration. The configuration constraint ensures that at most one of the variables conf_e corresponding to the events in E' is assigned the value 1, i.e., $1 \geq |\{e \in E' \mid \mathsf{conf}_e = 1\}| = \bigvee_{e \in E'} \mathsf{conf}_e$, and so a single $\vee$-gate is sufficient to count the number of variables assigned the value 1.

Definition 1 (#-cluster). *A set of events $E' \subseteq E \setminus E_{cut}$ is a* #-cluster *if for all distinct events $e, f \in E'$, $e\#f$.*

Thus the non-cut-off events of the prefix are partitioned into #-clusters, then $\vee$-gates are used to count in each #-cluster the number of variables corresponding to its events and assigned the value 1, and a counter (hopefully, of a much smaller size) is used to count the number of outputs of these $\vee$-gates having the value 1. Since the translation of an $\vee$-gate into a Boolean expression is much smaller than the translation of a counter, one can expect reductions in the size of the resulting formula. For example, $\{\{e_1\}, \{e_2\}, \{e_3, e_5\}, \{e_4, e_6\}, \{e_7, e_8\}\}$ is a possible partition into #-clusters of the non-cut-off events of the prefix shown in Fig. 1(b).

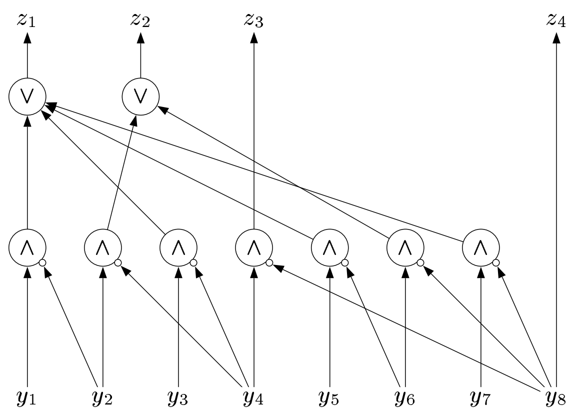
When partitioning the non-cut-off events of the prefix into #-clusters, it is advantageous to make the number of such #-clusters as small as possible. (When the number of #-clusters is large, the size of the counter grows; in particular, for the trivial partition with each event forming its own #-cluster the translation degrades to the one described in the previous section.) Thus one can formulate an optimisation problem of partitioning the non-cut-off events of a prefix into the smallest number of #-clusters. Unfortunately, a decision version of this problem turns out to be NP-complete.

Proposition 1 (NP-completeness of the Partition into #-clusters problem). *Given an unfolding prefix π and a $k \in \mathbb{N}$, the problem of deciding whether the set of non-cut-off events of π can be partitioned into at most k #-clusters is NP-complete.*

The proof is by reduction from the *Partition into Cliques* problem, which is known to be NP-complete [3, Problem GT15], and can be found in [6].

When computing the shortest violation trace, one does not want to spend too much effort on building the threshold constraints, as the process of building them can easily become more time consuming then model checking itself. Therefore, in the actual implementation, a fast 'greedy' algorithm for partitioning the set of events into #-clusters was adopted, which is justifiable in the view of the above result. This algorithm is described in [6].

Exploiting the causality relation The method described above allowed for simplification of the threshold constraint by exploiting the conflict relation between the events in the prefix. It turns out that the causality relation can also be exploited to reduce the size of the translation even further.



$$z_i \overset{\mathrm{df}}{=} \bigvee_{j=0}^{\left\lceil \frac{k-2^{i-1}}{2^i} \right\rceil} y_{2^i(j+\frac{1}{2})} \wedge \neg y_{2^i(j+1)}$$
$$i \in \{1, \ldots, \lceil \log_2 k \rceil + 1\}$$
$$y_{k'} \overset{\mathrm{df}}{=} 0 \text{ if } k' > k$$

Fig. 5. An implementation of an eight-input counter with the values of inputs constrained to be in a non-increasing order.

Definition 2. *Let Cl and Cl' be two #-clusters. $Cl \ll Cl'$ if for each event $e' \in Cl'$ there exists an event $e \in Cl$ such that $e < e'$. A sequence of #-clusters $Cl_1 \ll Cl_2 \ll \cdots \ll Cl_k$ is called a* $\ll$-chain.

For example, $\{e_4, e_6\} \ll \{e_7, e_8\}$ is a $\ll$-chain of the prefix shown in Fig. 1(b).

It follows from this definition that if $Cl \ll Cl'$ and an event $e' \in Cl'$ belongs to a configuration C then some event $e \in Cl$ also belongs to C. Suppose $Cl_1 \ll Cl_2 \ll \cdots \ll Cl_k$ is a $\ll$-chain and $y_1, \ldots, y_k$ are the outputs of the $\vee$-gates corresponding to these #-clusters. The configuration constraint ensures that in any satisfying assignment *the sequence of values of $y_1, \ldots, y_k$ is non-increasing.* This allows one to count the number of ones among these values much more efficiently than by a counter described in the previous section. Indeed, the encoding of the inputs is very similar to the 1-hot encoding, which can be obtained from $y_1, \ldots, y_k$ as $\neg y_1, y_1 \wedge \neg y_2, y_2 \wedge \neg y_3, \ldots, y_{k-1} \wedge \neg y_k, y_k$ and subsequently converted into the binary code using an encoder. A somewhat smaller circuit is shown in Fig. 5.

Thus one can partition the acyclic directed graph $G_{\ll}$ corresponding to the $\ll$ relation on the #-clusters into $\ll$-chains, then build for each $\ll$-chain a circuit similar to the one shown in Fig. 5, and finally construct an adder tree similar to that in Fig. 4(c), but with the bottom layer comprised of the built counters rather than half-adders. The algorithm shown in Fig. 6 does this trying to balance the resulting adder tree. *ExtractMin*(Q) extracts and returns a pair $(c, m) \in Q$ (where c is a circuit and $m \in \mathbb{N}$ is the maximum value this circuit can output) with the minimum value of m, and *Add*(c_1, c_2) constructs a circuit which computes the sum of values computed by c_1 and c_2 (i.e., an adder is put 'on top' of c_1 and c_2). Note that Q is a priority queue and can be efficiently implemented as either a binary heap or by keeping a list of circuits for each m.

When partitioning $G_{\ll}$ into $\ll$-chains, it is advantageous to make the number of such $\ll$-chains as small as possible, in order to reduce the number of adders in the adder tree. Thus one can formulate an optimisation problem of partitioning

```
input : $Q$ — a non-empty set of pairs $(c, m)$, where $c$ is a circuit and $m \in \mathbb{N}$
output : $c$ — a circuit

while $|Q| > 1$ do
    $(c_1, m_1) \leftarrow \mathit{ExtractMin}(Q)$
    $(c_2, m_2) \leftarrow \mathit{ExtractMin}(Q)$
    $Q \leftarrow Q \cup \{(\mathit{Add}(c_1, c_2), m_1 + m_2)\}$

/* now $|Q|$=1 */
$(c, m) \leftarrow \mathit{ExtractMin}(Q)$
return c
```

Fig. 6. An algorithm for building a tree of adders.

$G_{\ll}$ into the smallest number of $\ll$-chains. This is essentially the well-known *minimum vertex-disjoint path cover* problem (zero-length paths comprising a single vertex are admissible).

This problem is NP-complete for general graphs, since checking the existence of a Hamiltonian path is equivalent to checking whether it is possible to cover the vertices of a given graph by a single vertex-disjoint path. Nevertheless, for acyclic graphs (note that $G_{\ll}$ is acyclic) it can be reduced to the maximum matching problem on a bipartite graph, and solved in polynomial time [4]. However, one should bear in mind that $G_{\ll}$ is given implicitly, and can be very large. (It is not uncommon to have an unfolding prefix with hundreds thousands events.) Therefore, using an exact algorithm for solving this problem might be either too memory demanding (if $G_{\ll}$ is built explicitly), or too slow due to the need of working with an implicitly represented graph (checking whether there is an arc between two vertices of $G_{\ll}$ is quite expensive in such a case, as one might have to traverse the whole prefix). Thus a fast 'greedy' algorithm for partitioning the set of #-clusters into $\ll$-chains has been designed. It is described in [6].

4 Experimental results

The proposed method has been tested with the zChaff SAT solver [8], and the popular set of deadlock checking benchmarks collected by J.C. Corbett [1] has been attempted. (For obvious reasons, only examples with deadlocks from this collection were used.) All the experiments were conducted on a PC with a Pentium^{TM} IV/2.8GHz processor and 512M RAM.

The experimental results are shown in Table 1, where the meaning of the columns is as follows (from left to right): the name of the problem; the number of non-cut-off events in the prefix; the lengths of the first computed and a shortest violation traces; the number of #-clusters and $\ll$-chains computed by the heuristic algorithms described in [6]; the size (the number of new variables, clauses and literals) of the translation of the counter circuit for the basic translation described in Section 2 and for the improved one described in Section 3; and the time taken by the SAT solver to compute the first violation trace and the time taken by the algorithm in Fig. 3 to compute a shortest violation trace using the basic and the improved translations of the counter.

Problem	Prefix	Trace		Partitions		Translation of counter						Time		
						Basic			Improved					
	$\lvert E \setminus E_{cut} \rvert$	1st	shtst	#-cl	$\ll$-ch	vars	cls	lits	vars	cls	lits	1st	Bas.	Imp.
Q	7229	75	21	179	25	28881	115479	375221	520	8781	26031	<1	3	1
Speed	1663	24	4	30	9	6620	26436	85832	98	1952	5806	<1	1	<1
Dac(6)	53	6	6	23	11	195	761	2437	72	279	833	<1	<1	<1
Dac(9)	95	9	9	35	17	359	1409	4527	116	460	1372	<1	<1	<1
Dac(12)	146	12	12	47	23	564	2236	7230	160	662	2000	<1	<1	<1
Dac(15)	206	43	15	59	29	802	3182	10292	205	864	2600	<1	<1	<1
Dp(6)	66	6	6	18	6	247	973	3135	55	222	628	<1	<1	<1
Dp(8)	120	8	8	24	8	461	1823	5885	75	341	987	<1	<1	<1
Dp(10)	190	10	10	30	10	737	2919	9431	96	475	1381	<1	<1	<1
Dp(12)	276	12	12	36	12	1082	4306	13954	119	635	1861	<1	<1	<1
Elev(1)	98	9	9	16	5	374	1478	4770	43	222	640	<1	<1	<1
Elev(2)	496	22	12	24	7	1960	7812	25336	65	685	2017	<1	<1	<1
Elev(3)	2266	30	15	32	9	9033	36095	117239	94	2549	7607	<1	<1	<1
Elev(4)	9598	23	18	40	11	38354	153366	498344	117	9950	29798	2	27	3
Hart(25)	101	26	26	76	26	385	1519	4897	218	826	2528	<1	<1	<1
Hart(50)	201	51	51	151	51	783	3109	10061	440	1684	5188	<1	<1	<1
Hart(75)	301	76	76	226	76	1180	4692	15196	666	2566	7942	<1	<1	<1
Hart(100)	401	101	101	301	101	1581	6299	20425	888	3424	10602	<1	<1	<1
Key(2)	454	52	42	103	18	1792	7140	23152	285	1309	3761	<1	<1	<1
Key(3)	4057	53	43	223	41	16194	64730	210284	680	6123	18051	<1	20	2
Key(4)	35905	65	44	407	82	143582	574286	1866352	1269	39797	118855	<1	548	224
Mmgt(1)	38	6	6	11	2	136	528	1686	25	98	250	<1	<1	<1
Mmgt(2)	385	8	8	26	7	1518	6050	19622	80	618	1806	<1	<1	<1
Mmgt(3)	3312	10	10	36	6	13217	52831	171631	98	3584	10658	<1	<1	<1
Mmgt(4)	25945	12	12	44	7	103741	414915	1348381	119	26273	78693	77	86	80
Sent(25)	176	34	3	40	3	684	2716	8790	69	370	1028	<1	<1	<1
Sent(50)	201	59	3	65	3	783	3109	10061	98	480	1302	<1	<1	<1
Sent(75)	226	84	3	90	3	883	3509	11361	123	579	1549	<1	<1	<1
Sent(100)	251	109	3	115	3	980	3888	12574	149	681	1803	<1	<1	<1

Table 1. Experimental results for deadlock checking.

The experiments show that in many cases the first computed violation trace was much longer than a shortest one, with the results for the Sent benchmarks being particularly impressive. This confirms that in practice computing shortest violation traces can indeed greatly facilitate the debugging process.

One can see that the number of #-clusters and $\ll$-chains is usually quite small compared to the number of non-cut-off events in the prefix, and thus the reduction in the size of the formula is quite significant. It is possible to evaluate the maximum reduction which can be achieved by the improved translation over the basic one as follows. In the ideal case, all the events in the prefix would be in conflict with each other, and so the counter circuit can be implemented as a single $\vee$-gate. Such an implementation results in one new variable (for the gate's output), $n+1$ clauses and $3n+1$ literals in the corresponding CNF formula, where $n = |E \setminus E_{cut}|$. The corresponding parameters for the basic translation are given in Section 2, and the improvement ratios for new variables, clauses and literals are $(4n - 2\log_2 n - 4)/1 \approx 4n$, $(16n - 10\log_2 n - 16)/(n+1) \approx 16$ and $(52n - 36\log_2 n - 52)/(3n+1) \approx 17\frac{1}{3}$, respectively. Thus the reduction ratio for variables can grow unboundedly with n, whereas for clauses and literals it is bounded by 16 and $17\frac{1}{3}$, respectively.

The improvement ratios for the benchmarks in Table 1 are plotted in Fig. 7. One can see that for the number of new variables, the reduction ratio indeed grows with the size of the prefix (though not as fast as in the ideal case), and is

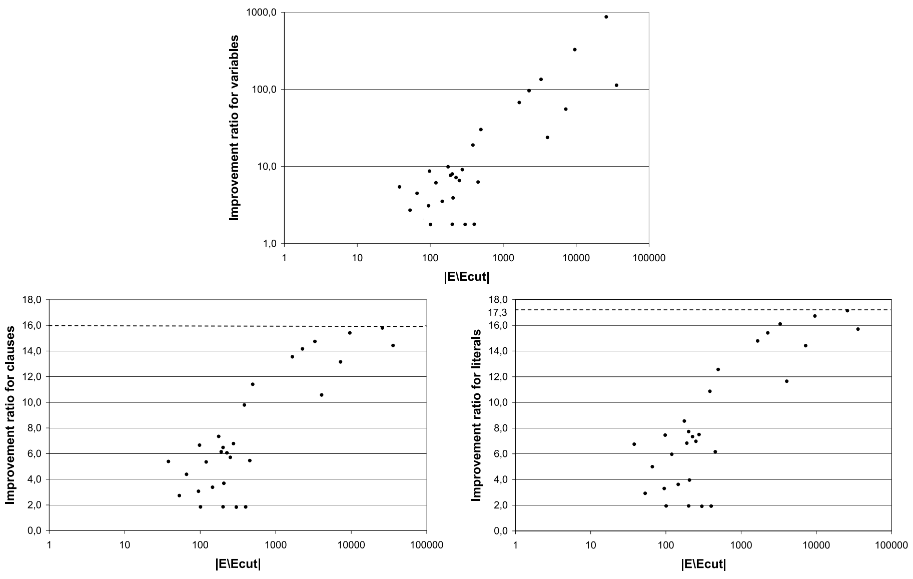


Fig. 7. Improvement ratios.

between two and three orders of magnitude for large benchmarks. For clauses and literals, the improvement rate also grows with the size of the prefix, and comes surprisingly close to the best possible ratio for large benchmarks. Moreover, it should be noted that since the improved translation uses a lot of multiple-input $\vee$-gates, the corresponding CNF formula has many clauses of length two, which makes the SAT instance easier for the solver.

The comparison of the running times of the algorithms shows that, except one test case, it was not too time-consuming to compute a shortest violation trace. (This is probably due to the fact that only a few benchmarks are large.) Moreover, the improved approach has a clear advantage over the basic one in terms of time. The only benchmark where computing the shortest violation trace by the improved method took significantly more time than just solving the original model checking problem was KEY(4). (Note that for MMGT(4) the increase in time was quite modest, which can be explained by the fact that the first computed violation trace was already optimal and very short.) In general, however, one can expect a significant increase in time when computing the shortest violation traces, due to the following phenomenon, related to *phase transition*. Let t^* be the length of the shortest violation trace. If t is significantly larger than t^*, adding the constraint $\mathit{Threshold}_t$ to the formula will exclude only a few satisfying assignments, and the resulting formula will not be much harder for the solver than the original one. On the other hand, if t is significantly smaller than t^*, adding $\mathit{Threshold}_t$ to the formula will yield an overconstrained SAT instance which usually can be quickly proven unsatisfiable. A hard situation can occur when t is close to t^*. In such a case, if the SAT instance is satisfiable, it often has only a small number of satisfying assignments (and thus such an assignment might be difficult to find), and if it is unsatisfiable, it might be hard to show this. The last part of Section 1 discusses how the impact of this phenomenon can be alleviated in practice.

5 Conclusions and future work

Although performed testing was limited in scope, one can draw some conclusions about the efficiency of the proposed approach. Computing shortest violation traces can facilitate the debugging process and save a lot of designer's time, since in many cases the first computed violation trace is much longer than a shortest one. According to the experimental results, for large problem instances it can reduce the number of new variables in the formula by two–three orders of magnitude, and achieve almost optimal reduction in the number of clauses and literals, i.e., the length of the CNF formula corresponding to the threshold constraint was surprisingly close to that for a single multiple-input $\vee$-gate!

The possible directions for future research include using a Boolean minimiser to derive short formulae not only for half-adder and full-adder cells but also for adders with a small number of inputs, and exploiting the structure of the prefix to reduce the size of other pseudo-Boolean constraints encountered when dealing with various model checking problems.

Acknowledgements The author would like to thank Keijo Heljanko for fruitful discussions. This research was supported by an EC IST grant 511599 (Rodin).

References

1. J. C. Corbett: Evaluating Deadlock Detection Methods for Concurrent Software. *IEEE Transactions on Software Engineering* 22(3) (1996) 161–180.
2. J. Esparza: Decidability and Complexity of Petri Net Problems — an Introduction. *Lectures on Petri Nets I: Basic Models*. LNCS 1491 (1998) 374–428.
3. M. Garey and D. Johnson: *Computers and Intractability — A Guide to the Theory of NP-completeness*. Freeman (1979).
4. J. E. Hopcroft and R. M. Karp: An $n^{5/2}$ Algorithm for Maximum Matching in Bipartite Graphs. *SIAM Journal on Computing* 2(4) (1973) 225–231.
5. V. Khomenko: *Model Checking Based on Prefixes of Petri Net Unfoldings*. School of Comp. Sci., Univ. of Newcastle (2003).
6. V. Khomenko: Computing Shortest Violation Traces in Model Checking Based on Petri Net Unfoldings and SAT. TRep. CS-TR-841, School of Comp. Sci., Univ. of Newcastle (2004). URL: `http://homepages.cs.ncl.ac.uk/victor.khomenko/home.formal/papers/papers.html`
7. K. L. McMillan: Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits. Proc. of *CAV'1992*, LNCS 663 (1992) 164–174.
8. S. Moskewicz, C. Madigan, Y. Zhao, L. Zhang and S. Malik: Chaff: Engineering an Efficient SAT Solver. Proc. of *DAC'2001*, ASME Techn. Publ. (2001) 530–535.
9. I. Wegener: *The Complexity of Boolean Functions*. Wiley-Teubner Series in Computer Science (1987).
10. L. Zhang and S. Malik: The Quest for Efficient Boolean Satisfiability Solvers. Proc. of *CAV'2002*, E. Brinksma and K. G. Larsen (Eds.). LNCS 2404 (2002) 582–595.