Efficient Strategies to Compute Invariants, Bounds and Stable Places of Petri nets

Yann Thierry-Mieg

Efficient Strategies to Compute Invariants, Bounds and Stable Places of Petri nets

Yann Thierry-Mieg[1,†]

[1]Sorbonne Université, CNRS, LIP6, F-75005 Paris, France

Abstract

This paper presents a set of strategies to solve global behavioral properties of Petri nets. We introduce efficient methods to compute structural invariants and bounds, and to decide presence of stable places for both PT nets and colored nets.

Our focus is on proposing strategies that scale to very large nets, even if precision is degraded or the test is only a semi-decision (e.g. a sufficient condition). These strategies form the decision procedures of ITS-Tools, that has won the annual model-checking competition MCC in these categories in both 2021 and 2022.

Keywords

Petri Nets, Model Checking, Structural properties

1. Introduction

The model-checking competition MCC [1] is an annual event held since 2011 comparing verification tools over a large set of Petri nets taken from diverse literature. The 2020 edition of the contest introduced a new category called "GlobalProperties" that contains five examinations :

- One Safe : in any reachable state, are all place markings bounded by at most one token?
- StableMarking : is there at least one place whose marking never evolves?
- Reachability of a Deadlock : are there reachable states where no transition can be fired?
- Quasi-Liveness : is there a way to fire any given transition starting from the initial state?
- Liveness : is there a way to fire any given transition starting from any given reachable state?

We are additionally interested in the "Upper Bounds" examination where we must compute the maximum number of tokens that can mark a given place of the net.

In this paper we propose integrated solutions to tackle some of these problems more efficiently. While most of these queries could be treated by simply translating them to a set of (reachability) assertions we try to avoid this step or at least reduce the number of queries needed to solve a given question. We also focus on providing a full decision procedure that starts with very simple and low complexity tests for sufficient conditions before escalating to incrementally harder tests.

These strategies have not been described elsewhere and form the basis of the decision procedures of ITS-Tools [2] that has been consistently winning the contest for these examinations since 2021.

PNSE'23: Petri Nets for Software Engineering, June 27, 2023, Lisbon, PT

first.last@lip6.fr (Y. Thierry-Mieg)

0000-0001-7775-1978 (Y. Thierry-Mieg)

The structure of the paper is the following, we first set the basic definitions and notations used in the paper Then for each property of interest, we present in detail a series of structural or low complexity sufficient or necessary condition that can be tested. If these preliminary strategies cannot conclude, we use a more complete model-checker but we limit as much as possible the number of queries that we pose to the model-checker. We discuss computation of invariants in section 3, then computation of bounds on place markings in section 4. This leads to decision procedures for the one safe property in section 5. Finally we present strategies to decide the stable marking property, i.e. are some place marking constant, in section 6.

2. Definitions

We first recall the definitions of a Petri net, then we informally introduce the notion of colored net (also called "High Level Petri Net") with its unfolding and skeleton.

2.0.1. Petri net syntax and semantics

Definition 1. *Structure. A Petri net* $N = \langle \mathcal{P}, \mathcal{T}, \mathcal{W}_-, \mathcal{W}_+, m_0 \rangle$ *is a tuple where* $\mathcal{P}$ *is the set of places,* $\mathcal{T}$ *is the set of transitions,* $\mathcal{W}_- : \mathcal{P} \times \mathcal{T} \mapsto \mathbb{N}$ *and* $\mathcal{W}_+ : \mathcal{P} \times \mathcal{T} \mapsto \mathbb{N}$ *represent the pre and post incidence matrices, and* $m_0 : \mathcal{P} \mapsto \mathbb{N}$ *is the initial marking.*

Notation: We use p (resp. t) to designate a place (resp. transition) or its index dependent on the context. We let markings m be manipulated as vectors of natural numbers with $|\mathcal{P}|$ entries. We let $\mathcal{W}_-(t)$ and $\mathcal{W}_+(t)$ for any given transition t represent vectors with $|\mathcal{P}|$ entries. $\mathcal{W}_-^\top, \mathcal{W}_+^\top$ are the transposed flow matrices, where an entry $\mathcal{W}_-^\top(p)$ is a vector of $|\mathcal{T}|$ entries. We denote $\mathcal{W}_e = \mathcal{W}_+ - \mathcal{W}_-$ the integer matrix representing transition *effects*.

In vector spaces, we use $v \geq v'$ to denote $\forall i, v(i) \geq v'(i)$, and offer sum $v + v'$ and multiplication of a vector by a scalar $k \cdot v$ for scalar k with usual element-wise definitions.

We denote $\bullet n$ (resp. $n\bullet$) the pre set (resp. post set) of a node n (place or transition). E.g. for a transition t its pre set is $\bullet t = \{p \in \mathcal{P} \mid \mathcal{W}_-(p,t) > 0\}$. A marking m is said to *enable* a transition t if and only if $m \geq \mathcal{W}_-(t)$. A transition t is said to *read* from place p if $\mathcal{W}_-(p,t) > 0 \wedge \mathcal{W}_e(p,t) = 0$.

Definition 2. *Semantics. The semantics of a Petri net are given by the firing rule* $\xrightarrow{t}$ *that relates pairs of markings: in any marking* $m \in \mathbb{N}^{|\mathcal{P}|}$*, if* $t \in \mathcal{T}$ *satisfies* $m \geq \mathcal{W}_-(t)$*, then* $m \xrightarrow{t} m'$ *with* $m' = m + \mathcal{W}_+(t) - \mathcal{W}_-(t)$*. The reachable set* $\mathcal{R}$ *is inductively defined as the smallest subset of* $\mathbb{N}^{|\mathcal{P}|}$ *satisfying* $m_0 \in \mathcal{R}$*, and* $\forall t \in \mathcal{T}, \forall m \in \mathcal{R}, m \xrightarrow{t} m' \Rightarrow m' \in \mathcal{R}$.

2.0.2. Colored Petri net syntax and semantics

A colored or high-level Petri net is similar to a Petri net structurally, but each place p now has a *domain* $d(p)$ and place markings are a multiset (or bag) over this domain. Similarly, an arc touching a place will bear a *color function* that itself resolves to a multiset over the domain of the place. Finally, transitions come equipped with a *guard* that can forbid or enable the transition depending on the type of tokens present in the connected places.

We avoid providing a formal definition of a colored net here, we recommend [3] for a formal and recent presentation of the subject. In this work, we study colored nets by inspecting their unfolding, a Petri net whose behavior exactly reflects the semantics of the colored net. However, unfolding a net can be explosive [3], so that we first try to answer queries using the skeleton net [4]. This is a Petri net that is obtained by basically forgetting about place domains and guards, and replacing any multiset expression over a domain simply by the cardinality of the multiset.

This operation has a very low complexity, but yields a net whose behaviors are a strict superset of the behaviors of the original colored net : the skeleton net can do anything that the original net could do, and more. Hence if the skeleton net is unable to reach a certain behavior, the original net cannot reach it either.

3. Computing invariants

The process of calculating the structural invariants of a Petri net, also known as flows, is based on an analysis of the incidence matrix (effects) of a net. The strategy is well known, it essentially consists in applying Farkas's lemma to an extended matrix [5], and many extensions and revisited versions have been proposed [6]. Its main strength is that it is a cheap structural approach with complexity related to the size of the net (not its state space) but that still provides a decent approximation of all reachable states.

This section presents the precise variant of this algorithm used within ITS-Tools. Our implementation is in plain Java, and is based on source-code we took and adapted from the APT [7] framework (with D-M. Borde and M. Gieseking listed as authors), and where the code further traces the origins to the PIPE [8] tool. We first present the algorithm then compare it to the solution offered by the Tina [9] package.

3.1. Data structures and setup

To store a vector v we use a sparse array data structure, containing two integer arrays *indexes* and *values* both of size k the number of non zero entries in v. Indexes are maintained sorted, values can never be 0. For instance vector $v = [0,0,6,0,0,4]$ would be stored as $indexes = [2,5], values = [6,4], k = 2$. Iterations of non zero entries can be done in $O(k)$, and computing the number of non zero entries is $O(1)$. However, lookups with an index are logarithmic to k, arbitrary insertions are $O(k)$, but appending is $O(1)$ (without reallocation). Multiplication by a scalar as well as summing two sparse arrays are also sparse operations in $O(k)$.

To store matrices we use a dynamically allocated array (ArrayList in Java) of sparse arrays representing the columns of the matrix. So this structure is "semi sparse", each column has an entry in the matrix even if it is empty. Transposition is possible with complexity proportional to the number of non zero entries (exploiting the fact that appending to a sparse array is $O(1)$).

We initialize the algorithm by computing the matrix $\mathcal{W}'_e$:

- Let $\mathcal{W}_e$ designate the effects matrix where columns are transitions and places are rows
- We first build a set from the normalized columns of $\mathcal{W}_e$; a set so elements are unique, and normalized by dividing all entries in the vector by the greatest common divisor of its

entries. These operations will discard transitions identical up to a multiplicative factor on arc inscriptions, as well as fuse any two transitions that differ only by "read" arcs, i.e. that have the same effect but different enabling conditions. Applying this normalization step is a new idea to the best of our knowledge.
- We then rebuild a matrix from this set (column order being unimportant), and finally transpose it to obtain the input $\mathcal{W}'_e$ of the next step where rows are transitions and columns are places.
- We then compute for each row of the $\mathcal{W}'_e$ matrix the set of indexes that correspond to strictly positive and strictly negative entries. This data structure is stored in a variable pm and updated during the computation. This structure is important to stay in complexity related to non zero entries when working with a row index, so that we can directly find the columns that contain non zero values in this row.

3.2. Main loop

Our goal in the next step is to empty the matrix $\mathcal{W}'_e$, by replacing columns in the matrix by linear combinations of other columns until we converge to a matrix only containing zeroes. Basically we choose a row r with some non zero entries in it, we then pick a column k where the row has a non zero entry $r[k]$, and replace all other columns j that have a non zero entry in the row with a linear combination of columns k and j with coefficients chosen so that $r[j]$ is set to 0. We then discard column k.

After each such step, another row in the matrix is nullified, so that convergence is ensured in a polynomial number of steps. Progress is tracked by applying the same operations to an identity matrix, from which invariants (if any) can be read directly at the end of the procedure. This setting is classical, but we focus on only using *sparse* operations.

Let I designate the identity matrix with dimension equal to the number of columns in $\mathcal{W}'_e$ (i.e. the number of places in the net). We initialize a square matrix $B = I$.

We then iterate the following procedure until the $\mathcal{W}'_e$ matrix is empty :

- Find a row r in the matrix such that exactly one entry is strictly negative or positive. We use pm to test this. If such a row exists :
  - Let k designate the index of the single strictly positive (resp. negative) entry in r. Then for all j, $j \neq k$, such that $r[j] < 0$ (resp. strictly positive),
    - First compute $g = gcd(|r[k]|, |r[j]|)$, then the coefficients $\alpha_j = |r[j]|/g$ and $\alpha_k = |r[k]|/g$. We then replace column j of $\mathcal{W}'_e$ by column $c = \alpha_k \cdot \mathcal{W}'_e[j] + \alpha_j \cdot \mathcal{W}'_e[k]$. While computing c, we can compute indexes of rows r' where $\mathcal{W}'_e(j)[r'] \neq c[r']$, and use this information to (sparsely) update pm.
    - We then also replace the column $B(j)$ of index j in B by $B(j) = \alpha_k \cdot B(j) + \alpha_j \cdot B(k)$.
  - Finally we clear the column k of $\mathcal{W}'_e$, updating pm (sparsely, based on non zero values of $\mathcal{W}'_e(k)$), and clear column k of B. Note we use the term "clear" deliberately here, it is much preferable to set all entries to zero in a sparse data structure than actually removing a column (and requiring to reindex elements with greater indices

non sparsely in pm). Note that empty columns are irrelevant with respect to our algorithm, and induce almost no extra cost with sparse data structures.

- If no such "exactly one entry is strictly negative or positive" row exists in $\mathcal{W}'_e$, we look for a row r in $\mathcal{W}'_e$ such that there exists an index k such that $r[k] \neq 0$.
  - While technically any such entry is appropriate, heuristically we want to choose a position that minimizes subsequent iterations of the algorithm. We currently orient this choice by looking for the column with index k in $\mathcal{W}'_e$ that has the least non zero entries (reducing the number of subsequent operations, and fast to test with sparse structures), and in case of equality by taking the column with smallest summed absolute value $\sum_i |\mathcal{W}'_e(k)|$ (reducing the risk of coefficient blowup).
  - Given k, we let r designate a row of $\mathcal{W}'_e$ such that $r[k] ! = 0$. We can choose any such row without impacting performance of the algorithm, so we simply take the first non zero entry in column of index k.
  - Using pm to identify them we then iterate over all other non zero entries of r : let j designate such an entry ($j \neq k, r[j] \neq 0$), and $g = gcd(r[j], r[k])$ designate the greatest common divisor of these entries.
    - Let $\alpha_k = |r[k]|/g$. Let $\alpha_j = r[j]/g$ if $r[j]$ and $r[k]$ have opposite sign, or $\alpha_j = -r[j]/g$ otherwise. We then replace column j of $\mathcal{W}'_e$ by a new column $c = \alpha_k \cdot \mathcal{W}'_e(j) + \alpha_j \cdot \mathcal{W}'_e(k)$. During this product, and similarly to above we should (sparsely) update the pm data structure.
    - We then replace the column $B(j)$ of index j in B by $B(j) = \alpha_k \cdot B(j) + \alpha_j \cdot B(k)$.
  - Finally we clear column k from both $\mathcal{W}'_e$ and B (and update pm).
- Note that we must iterate the procedure to a fixpoint. We end the procedure when $\mathcal{W}'_e$ is empty hence none of the two above rules apply. We always iterate the search for an appropriate row for the first test by iterating the rows in pm starting from *the last row index where a rule applied* rather than starting from row 0, as this avoids repeating redundant tests in most cases (and iterating elements in pm is not sparse).

At this stage, the $\mathcal{W}'_e$ matrix has been entirely emptied, and we can find the invariants in the B matrix. We can post process the columns of B by dropping empty columns, normalizing columns (by dividing by the gcd of its entries, and possibly multiplying the column by -1 if all entries are negative), and filtering any duplicate columns.

This algorithm produces a generative basis of the P-flows of the net, but can be applied to $\mathcal{W}_e^\top$ if we wish to compute T-flows. This basis of flows can further be used to compute semi-flows [5], i.e. invariants where all coefficients are positive, that are often considered easier to read and interpret by humans. However, while the number of flows is guaranteed to be at most equal to the number of places in the net, the number of semi flows can be exponentially larger so that computing them could be a limiting factor in a decision procedure. Since in our approach flows are mostly used as a very rough approximation of reachable states that is then analyzed with an SMT solver (that does not really mind negative coefficients), we prefer to stop the computation at this step.

Notes: The idea of first looking for rows with a single positive/negative entry comes from PIPE [5], it avoids introduction of negative coefficients unless they are really needed and favors

positive solutions (semi-flows) over generalized flows. Overall, it is critically important to ensure all operations are sparse operations, with complexity related to number of non zero entries, this is the main strength of this implementation. Incidence matrices of Petri nets are notably sparse, but can be very large (think thousands of places, tens of thousands of transitions but on average only a few arcs per transition). The normalization of α coefficients using the gcd during the computation is important to avoid multiplicative growth of entries in $\mathcal{W}_e$ that could lead to an integer overflow pretty easily and is original to the best of our knowledge. The heuristic we use to choose a pivot in the second part is also original to the best of our knowledge.

3.3. Performance comparison

In this section we evaluate the performance of our implementation of this algorithm. We chose as comparison the tool Tina [9] as it is a mature, efficient and well maintained package that proposes this functionality, when some implementations that seem promising such as [6] are no longer available (if they ever were). Tina proposes a native implementation (probably implemented in StandardML, but source is not public) and an implementation that relies on 4ti2 [10] an efficient C++ package dedicated to linear algebra to compute the solution. This second solution is more efficient than the native implementation. We did not compare to either PIPE [8] or APT [7] implementations despite our code originating from them since they are not sparse implementations and really cannot compete.

The following performance evaluation [1] was performed on a Linux machine intel Core 7. We used the full set of 1384 PT models from the 2022 edition of the model-checking contest and computed P flows and T flows with both tools. The following table as well as Figure 3 summarize the results. The column "Overflow" corresponds to integer overflowing in the representation of the coefficients in the matrix. "Timeout" corresponds to exceeding the 120 seconds time limit, and "Memory" is runs exceeding 16GB of RAM.

Tool	Success	Overflow	Timeout	Memory
ITS-Tools	1355	20	9	0
Tina 4ti2	1215	1	157	7
Tina	948	145	289	1

While in a few cases (16 overall) we meet integer overflow when Tina can produce a solution (particularly with 4ti2), problems ITS-Tools can't solve within one second are mostly unsolvable by Tina, and our implementation is extremely memory efficient (peaking at 3GB) despite being pure Java. Our implementation thus outperforms Tina [9] ("struct" component) even using "4ti2" library [10], and scales to very large nets (up to 143908 places, up to 373236 transitions and up to 8944506 arcs for the largest solved instances). The implementation is freely available as open source code [2].

[1] To help reproduction of these experiments, we built a repository with the raw data and all useful commands, see https://github.com/yanntm/InvariantPerformance

[2] Search for file "InvariantCalculator" in the https://github.com/lip6/ITSTools repository.

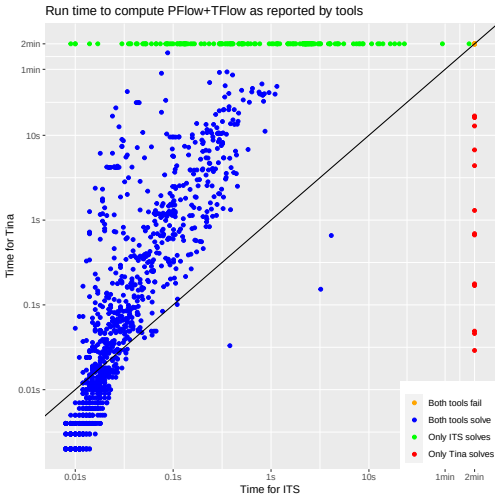

Figure 1: Comparison in solution time

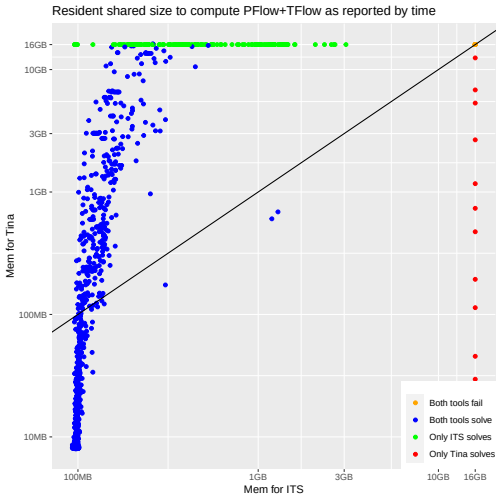

Figure 2: Comparison on memory occupation

Figure 3: Experiments on the MCC'2022 models using ITS-Tools and Tina (with 4ti2). Each point is a model, with horizontally time/memory consumption for ITS-Tools, and vertically consumption by Tina plotted using Log/Log scale. We plot any timeout or memory overflow at 2 minutes in time and $16GB$ in RAM. ITS-Tools is both much faster and consumes much less memory, despite the startup penalty due to the Java runtime requiring $\approx 100MB$ RAM.

4. Computing Bounds

The goal of this task is to compute the maximum number of tokens in a place or group of places that can actually be reached. Our approach consists in computing both a lower bound Max_{seen} that represents the highest number of tokens actually confirmed to have been seen in this place or group of places and an upper bound Max_{struct} that represents a structural maximum that the marking cannot exceed. When both values are equal, we can conclude, this value is the bound on this place marking.

For instance, suppose we must compute the bounds on expression $p_0 + p_1$. Suppose that in the initial marking, $p_0 = 1$ and $p_1 = 0$, so that Max_{seen} is 1. If we can compute an invariant that states $p_0 + p_1 + p_2 = 1$, we can deduce that Max_{struct} is also 1, so that the bound of $p_0 + p_1$ is 1.

For the lower bound Max_{seen}, we can simply explore (parts of) the state space, remembering the highest value encountered. For the upper bound Max_{struct}, we seek to compute structural bounds if we can, rather than relying on an exhaustive exploration of the state space.

4.1. Computing rough structural upper bounds using P flows.

As a first step, we compute rough upper bounds on reachable markings using P flows. We first compute the P-flows, using the algorithm presented in section 3.

The solution can naturally contain some positive invariants (semi-flows) that we integrate first. Such an invariant is of the form : $\alpha_0 p_0 + \alpha_1 p_1 + \ldots = K$ with alpha coefficients and K being natural integers in $\mathbb{N}$. Since at best all p_i variables are 0 except one of them, we can compute an

approximate upper bound on each place marking p_i with a non zero coefficient α_i in the invariant as K/α_i where / designates Euclidean division. For instance, suppose we have the invariant $2 \cdot p_0 + p_3 = 5$, we can deduce that $p_0 \leq 2$ and $p_3 \leq 5$.

Now that at least some variables have an upper bound, we can look at generalized flows with both positive and negative coefficients. We write them under the form $(\alpha_0 p_0 + \alpha_1 p_1 + \ldots) - (\beta_0 p_0 + \beta_1 p_1 + \ldots) = K$ where α and β coefficients are natural integers, and for any given place p_i, one of α_i or β_i at least is zero, and K is a relative integer. This presentation highlights a positive term (α_i) and a negative term (β_i). If either of these two terms is bounded, we can now bound the other one. For instance, suppose we have the invariant $p_0 - p_1 = 2$, and that we have the bound $p_1 \leq 1$ already, we can deduce that $p_0 - 1 \leq 2$ hence $p_0 \leq 3$. The same invariant $p_0 - p_1 = 2$ can also be written $p_1 = p_0 - 2$, so if we know that $p_0 \leq 5$ we can deduce that $p_1 \leq 3$. We iterate the procedure on generalized flows until no new bounds are detected since results might otherwise depend on the order in which the invariants are inspected.

If for instance all the places of the net are covered by simple semi-flows $p_0 + p_1 + p_2 = 1$, $p_3 + p_4 + p_5 = 1 \ldots$ this "rough" strategy is still enough to conclude correctly that the net is one safe. Note however that the computations are very rough, for instance suppose we have invariants $p_0 + p_1 = 1$ and $p_0 + p_1 - p_2 = 0$. The first invariant gives us the bounds $p_0 \leq 1$ and $p_1 \leq 1$, the second invariant however will give $p_2 \leq 2$ because we approximate both p_0 and p_1 using their current bound (despite having a tighter bound on $p_0 + p_1$). Of course a more powerful approach, e.g. an ILP or SMT solver would be able to better exploit this kind of tight bounds, however this "rough" strategy is extremely fast and scales just as well as invariant computation does (so it scales *much better* than delegating problems to an ILP/SMT solver or doing any kind of actual model-checking).

4.2. Refining structural upper bounds with SMT

As described in [11], ITS-Tools uses a series of incrementally more complex constraints to structurally approximate the state space with an SMT solver (we use Microsoft Z3[12]).

In this process, the SMT solver works with an over-approximation of the state space so that states unreachable according to SMT are definitely unreachable, but SMT may believe some states are reachable when it is is not the case.

When we query the SMT solver, we ask it to exhibit a "reachable" marking satisfying a given predicate p. We thus either get a trusted "UNSAT" answer, that means such a situation is not possible, or we get a "SAT" answer that cannot be fully trusted (but that we can try to replay, see [11] for details).

The strategy to define "reachable" consists in successive approximations that are progressively more constraining. We start with reals (where the complexity of any linear algebra is much lower) and only try natural solutions in a second step. We introduce constraints and additional variables to support those constraints incrementally, starting with flows, then the state equation, then trap constraints. . . as explained in [11] in details.

Suppose we have computed a lower bound Max_{seen} on the marking of a place or group of places. We ask the SMT solver to test whether a state with *strictly more* tokens than Max_{seen} in those places is "reachable". If we get "UNSAT", we can conclude, this maximum seen bound cannot be exceeded, it is the bound on this expression.

When we get "SAT", we can try to replay the solution provided by the solver to confirm it. In this process, we exploit the fact that Z3 has an optimizing component, so that we ask the solver to maximize the value of the target expression when it reaches a SAT verdict. This lets the example exhibited by the solver (if we can replay it) be potentially a run that exhibits the true bound, so that in the next iteration when Max_{seen} has this value and we ask the solver to produce a state with a larger value, we actually get UNSAT and can conclude.

4.3. A full workflow for Upper Bounds

Our full strategy to compute upper bounds thus consists in the following steps :

- For colored nets, we can start by computing Max_{struct} using the skeleton net and the rough approximation of section 4.1 and/or the more refined one provided by SMT in section 4.2. This is correct since the behavior of the skeleton net is an over-approximation of the behavior of the net. We then unfold the net and proceed with the same algorithm as for PT nets.
- For any kind of net, we initialize Max_{seen} with the value of expressions in the initial marking. We also compute the P flows of the net and initialize Max_{struct} using the rough structural bounds of section 4.1.
- We then iterate the following steps until a solution is reached (or we time out) :
  - If $Max_{struct} = Max_{seen}$ conclude
  - Try to increase Max_{seen} by exploring (parts of) the state space. We can use pseudo-random or directed walks for instance. We do not need to be exhaustive, Max_{seen} is always an under-approximation of the bound we are looking for. So any state space exploration is fine, ITS-Tools typically runs a decision diagram engine, the LTSMin engine with partial-order reduction [13] as well as pseudo-random walk(s) [11] typically for a few seconds. If any of these strategies can build the whole state space, of course we can conclude.
  - Test if SMT believes that Max_{seen} can be exceeded. If not, conclude, else try to replay the solution.
  - Apply structural reduction rules preserving reachability [11] to the net such as agglomerations, detection of constant places, of dead or redundant transitions... yielding a smaller net for the next iteration. One rule that is specific to this process is when we are looking for a bound on a single place p and that $Max_{struct} = 1$ (hence $Max_{seen} = 0$ or we would already have concluded), we simply discard all transitions consuming from p.
- This whole strategy can complement a model-checking run that might conclude simply by exploring all states of the system; however in many cases where full model-checking is not possible due to state space explosion, our procedure still can conclude. In the worst case, we do detect some bounds even if not all, and alleviate any subsequent work for a more exhaustive approach.

4.4. Performance comparison

The "UpperBounds" category of the MCC consists precisely in this examination. Figure 6) is a performance comparison taken from the latest 2022 edition of the contest. It compares ITS-Tools

that obtained the gold medal in the category with a total of 23388 points against second place finisher Tapaal that gathered 21687 points (roughly 8% less points). The third place finisher GreatSPN using only exhaustive methods with advanced decision diagram technology [14] obtained 13274 points.

While we can observe a large blob of relatively easy answers computed within 10 seconds by Tapaal and that ITS-Tools can take up to two minutes to solve, there is also significant number of points against the right border, corresponding to timeouts of Tapaal when ITS-Tools could answer within a few minutes.

While our algorithm is very effective in most cases, our decision procedure cannot conclude when dealing with unbounded expressions where we should answer $+\infty$ (since Max_{seen} will never reach this bound with the mechanism we propose to increase it). While they were exceedingly rare originally, the number of unbounded models in the MCC benchmark has been growing steadily, so that dealing with such models is clearly a direction for improvement.

5. One Safe

The goal of this examination is to prove that a net is one safe, i.e. in all reachable markings, all place markings are bounded by one. For colored nets the definition is the same, at most one token (all colors confused) per place is allowed.

5.1. Elementary tests

First we check if the initial state satisfies the property. If not, we can conclude.

Note that this test is conclusive for many models, particularly colored ones where the requirement of having a single token in a colored place is pretty extreme (there is some debate on the definition that was adopted in the MCC, the "color safe" property where there are never more than one token of each color in a given place is perhaps more interesting), and ones with large initial markings (e.g. scaleable models).

The next test consists simply in checking whether the input file contains a relevant NUPN section [15], i.e. a decomposition of the net into simple interacting automata. The format includes a tag "safe" that is true if the net is guaranteed to be one safe. This additional decomposition into NUPN used to be provided only with a few models in a "tool specific" section of the PNML model as an artefact of their generation process, but currently many models of the benchmark bear this information, it was apparently computed a posteriori by the model maintainers [15] and added to many of the benchmark models in 2020.

5.2. Reducing the amount of queries

A basic approach given a decision procedure that can prove an invariant is to simply place one assert on each place of the net and try to prove this as one single big invariant $\bigvee_{p\in\mathcal{P}} p \leq 1$ or as a set of invariant queries $\forall p \in \mathcal{P}, p \leq 1$. However this approach can scale poorly when the number of places in the net grows due to the excessive size of the query for the "one big invariant" approach or to the number of queries in the set of queries approach.

In the spirit of gradually treating the problem, we do think of the problem as a set of invariants that we need to prove. Hence each one we can treat reduces the amount of work remaining for downstream procedures.

Our next step consists in reusing the algorithm proposed to compute upper bounds of section 4.1. We try to prove that $Max_{struct} = 1$ for as many places as we can. Even if this bound is not reachable (i.e. we have not computed Max_{seen}) we can still conclude the place is one safe. Places thus proved to be one safe can be removed from the remaining queries. This strategy is particularly effective on many nets whose places are covered by simple P semi flows.

Still in the spirit of reducing the amount of queries, we can look for places that cannot be fed (and are initially one safe). Such a place can only allow output transitions to fire a single time. Hence if these transitions are the single input of some place, that place is also one safe. More formally, let $P_s \subseteq \mathcal{P}$ designate an initially empty set of places, and $T_s \subseteq \mathcal{T}$ designate an initially empty set of transitions. We iterate to a fix point the following procedure :

- For any place p such that $\bullet p = \emptyset$, or $\bullet p = \{t\}$ a single transition and $t \in T_s$ and $\mathcal{W}_+(p,t) = 1$, then we add p to P_s : $P_s \leftarrow P_s \cup \{p\}$ and add all output transitions of p to T_s : $T_s \leftarrow T_s \cup p\bullet$.

At the end of the procedure, all places in P_s are known to be one safe.

Correctness comes from the fact that transitions in the set T_s can be fired at most once in any execution of the net, hence their output places can receive at most one token in any execution.

This is a low complexity exploration of the structure of the net, that bypasses actual state space exploration. It works particularly well for models of workflows that do not have loops. Any place we deduce is one safe reduces the number of subsequent queries to a reachability engine.

5.3. A full workflow for One Safe

Our full strategy to compute whether a net is one safe thus consists in the following steps :

- Test initial conditions and the elementary conditions of section 5.1.
- Use the rules of section 5.2 to reduce the amount of queries
- If we have not yet concluded, delegate remaining queries to a reachability/invariant decision procedure, as a set of invariants to prove.

These cheap structural tests ahead of a more complete model-checking approach thus help our solution scale to larger models.

5.4. Performance evaluation

The "OneSafe" category of the MCC consists precisely in this examination. Figure 4) is a performance comparison taken from the latest 2022 edition of the contest. It compares ITS-Tools that won the category with a total of 25922 points against second place finisher Tapaal that gathered 24707 points, GreatSPN [14] obtained 13145 points (roughly consistent with the number of models for which full state space construction is feasible with decision diagrams). The results on Figure 4 are much less ambiguous than the raw scores show, while the problem is not too hard to solve within the 30 minute deadline of the MCC for Tapaal (and most of the models it solves will be answered within 5 minutes), ITS-Tools treats practically all models within 10 seconds (with a few that take up to two minutes).

6. Stable Marking

The goal of the "Stable Marking" examination is to prove that there are no places whose marking is constant in all reachable states. This typically could indicate "dead code" or that parts of the net cannot be activated. It suffices to find a single "stable place" whose marking is a constant to answer the query.

6.1. Tests for Colored nets

For colored nets and prior to unfolding we try to detect that all outgoing arcs of a colored place to a transition t have a symmetric incoming arc from t inscribed with the same color function. This criterion for constant places in the net reflects a usage pattern in colored models of using complex place markings to encode discretized functions, that are "queried" but never actually updated. This can also be exploited during unfolding to reduce the size of the unfolded net. In any case such a place is stable letting us answer the query.

Still prior to unfolding, we can check if the skeleton net contains stable places (recursively using the same procedure, but it is much smaller). Since its behavior is a super set of the behaviors of the real unfolding, if a place is stable in the skeleton, it must also be stable in the unfolding. The reverse is not true, even if all places of the skeleton can be updated this does not imply that the unfolding satisfies the property.

Failing these two first tests we proceed to unfold the colored net.

6.2. Reducing the complexity

The first test on PT nets is to look for an empty row in the incidence matrix $\mathcal{W}_e$. More precisely we look for an empty column in $\mathcal{W}_e^\top$ given the sparse data structures we use to store matrices. This indicates a constant place, hence the net has "stable markings".

We then look for unmarked syphons (see [11], rule 10 in section 4), these are places that are not and never will be marked, hence they are stable and we can conclude.

The next step consists in building a graph representing "free" token flow. We let places be the vertices of the graph, and add an edge from place p to place p' iff. there exists a transition with $\bullet t = \{p\}, t\bullet = \{p'\}$ and arc weights one. Such "transfer" transitions could be an artifact resulting from model generation, but they also occur naturally in many manually built models. This graph is relatively simple, but we can exploit it to remove the burden of proving for many places that they are *not* constant.

We use two strategies, both based on computing suffixes in this graph.

Firstly, all places in the suffix of a place that is initially marked, as well as the place itself are *not stable*. Indeed, the tokens in the place can flow freely to any place "downstream".

Next we can take all remaining places that are not currently proved to be unstable but are connected in this graph, and compute a minimal set of places P such that all nodes in the graph are reachable from a node in P. This requires a small prefix search and computing the SCC of the graph, but the complexity of these operations is quite incomparable to any downstream model-checking treatment to prove invariants. More precisely, we look for a vertex that has no predecessor, add it to P and discard all places in its suffix (i.e. we assert that we have proven they

are *not* stable). When all nodes have a predecessor, only SCC and their suffix remain; we simply pick an arbitrary vertex belonging to an SCC, add it to P and discard all places in its suffix. The procedure is iterated until there are no more connected nodes in the graph.

We retain only places in P in the subsequent proof burden, and (optimistically) suppose that all places in their suffix (in the free token flow graph) have already been proved non stable. If one of the places in P is indeed stable, we will conclude instantly anyway, otherwise all places downstream can indeed be updated, hence the optimistic view we took is justified.

6.3. A full workflow for Stable Marking

Our full strategy to compute whether a net satisfies the stable marking property thus consists in the following steps :

- If the net is colored, check for constant colored places and try to prove the property using the skeleton prior to a full unfolding.
- Use the rules of section 6.2 to conclude if possible or at least reduce the amount of queries
- If we have not yet concluded, delegate remaining queries to a reachability/invariant decision procedure, as a set of invariants to prove of the form $m(p) \neq m_0(p)$ for all remaining places.

The structural tests help the solution scale, by reducing the burden of proof on subsequent steps. By eliminating the burden of proof for some places we also help the decision procedures in the reachability solver to bootstrap more effectively, particularly when they are sensitive to the *alphabet* of the property. For instance, places whose marking is used in a property cannot be reduced by structural reduction rules and typically neither can neighboring transitions be agglomerated.

6.4. Performance evaluation

The "StableMarking" category of the MCC consists precisely in this examination. Figure 5) is a performance comparison taken from the latest 2022 edition of the contest. It compares ITS-Tools that won the category with a total of 24405 points against second place finisher Tapaal that gathered 21560 points, GreatSPN [14] obtained 13145 points.

The results on Figure 5 overall advantage ITS-Tools particularly on harder models. There is a dense blob of points above the diagonal corresponding to Tapaal solving within one second while ITS-Tools takes as much as ten seconds. However as models become more difficult to solve, most points end up below the diagonal (and far from it, particularly in a log/log scale), and the relatively dense border to the right indicates many timeouts of Tapaal when ITS-Tools could conclude.

While part of these results come from the good performances of our reachability engine described in [11], the techniques presented in this section do contribute to the overall efficiency and help to deal with very large colored or generated models in particular.

7. Performances

In this section mostly consisting of figures we regroup performance plots taken from the website of the Model Checking Contest in its 2022 edition, see https://mcc.lip6.fr/2022/results.php.

These log/log scatter plots compare ITS-Tools that won each of these categories to Tapaal that obtained second place. Each point is a model, with horizontally time consumption for ITS-Tools, and vertically consumption by Tapaal. Points below the diagonal advantage ITS-Tools while points above it advantage Tapaal.

Timeout was 30 minutes on all examinations except upper bounds that is granted one hour. In total, 1617 model instances coming from 150 model families are represented. For upper bounds, each query additionally contains 16 bounds to compute on a given model, for the other plots a single answer is expected.

ITS-Tools is clearly more efficient than the runner up Tapaal on all plots, with OneSafe being the most blatant and Upper Bounds being the least clear cut.

8. Conclusion

Structural approaches to analyze Petri nets can allow to bypass the state space explosion that is the main enemy of model-checking in many cases. In this paper we have presented efficient strategies designed to answer some of the queries of the model-checking contest as well as computing invariants of a net.

These strategies are used by the state of the art verification engine ITS-Tools and are partly responsible for its success in the competition. While some "strategies" are seemingly trivial, this paper is a precise if informal description of otherwise unpublished algorithms that have been refined and tested over the huge benchmark of the MCC.

ITS-Tools is free open source under GPL, so please do feel free to look directly at the implementations. It is distributed from its home page https://ddd.lip6.fr and hosted on GitHub https://github.com/lip6/ITSTools.

Acknowledgments

We would like to thank Amine Benslimane and Sofiane Braneci who helped to integrate support for global properties into ITS-Tools during an internship in 2021.

References

[1] E. G. Amparore, B. Berthomieu, G. Ciardo, S. Dal-Zilio, F. Gallà, L. Hillah, F. Hulin-Hubard, P. G. Jensen, L. Jezequel, F. Kordon, D. L. Botlan, T. Liebke, J. Meijer, A. S. Miner, E. Paviot-Adet, J. Srba, Y. Thierry-Mieg, T. van Dijk, K. Wolf, Presentation of the 9th edition of the model checking contest, in: TACAS (3), volume 11429 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 50–68.

[2] Y. Thierry-Mieg, Symbolic model-checking using its-tools, in: TACAS, volume 9035 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 231–237.

[3] A. Bilgram, P. G. Jensen, T. Pedersen, J. Srba, J. H. Taankvist, Improvements in unfolding of colored petri nets, in: RP, volume 13035 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 69–84.

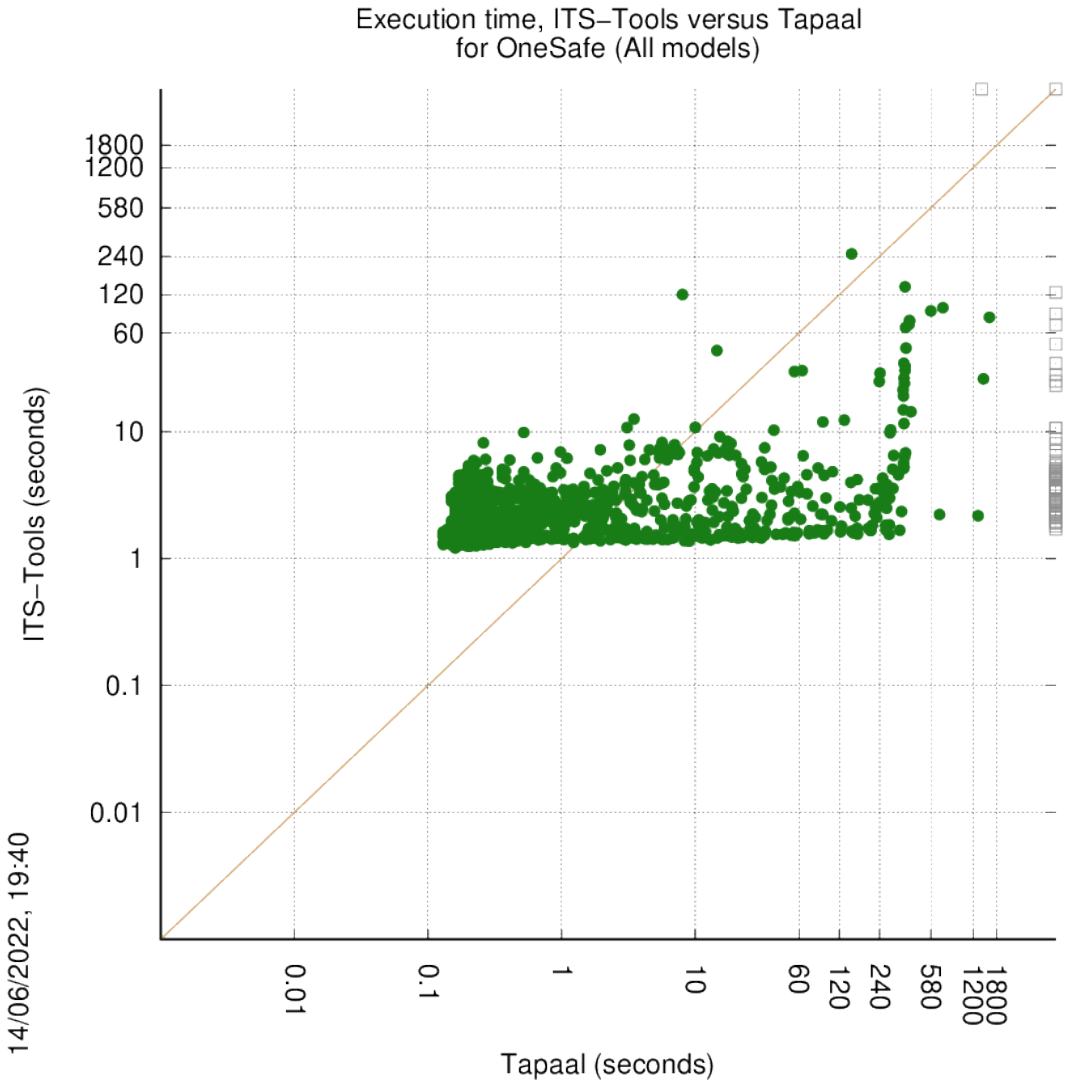

Figure 4: Results from MCC'2022 comparing ITS-Tools to Tapaal in time taken to solve the One Safe problems.

[4] S. Wallner, K. Wolf, Skeleton abstraction for universal temporal properties, Fundam. Informaticae 187 (2022) 245–272.

[5] M. D'Anna, S. Trigila, Concurrent system analysis using Petri nets: an optimized algorithm for finding net invariants, Computer Communications 11 (1988) 215–220.

[6] C. Law, B. Gwee, J. S. Chang, Fast and memory-efficient invariant computation of ordinary petri nets, IET Comput. Digit. Tech. 1 (2007) 612–624.

[7] E. Best, U. Schlachter, Analysis of Petri nets and transition systems, in: ICE, volume 189 of *EPTCS*, 2015, pp. 53–67.

[8] J. Bloom, C. Clark, C. Clifford, A. Duncan, H. Khan, M. Papantoniou, Platform inde-

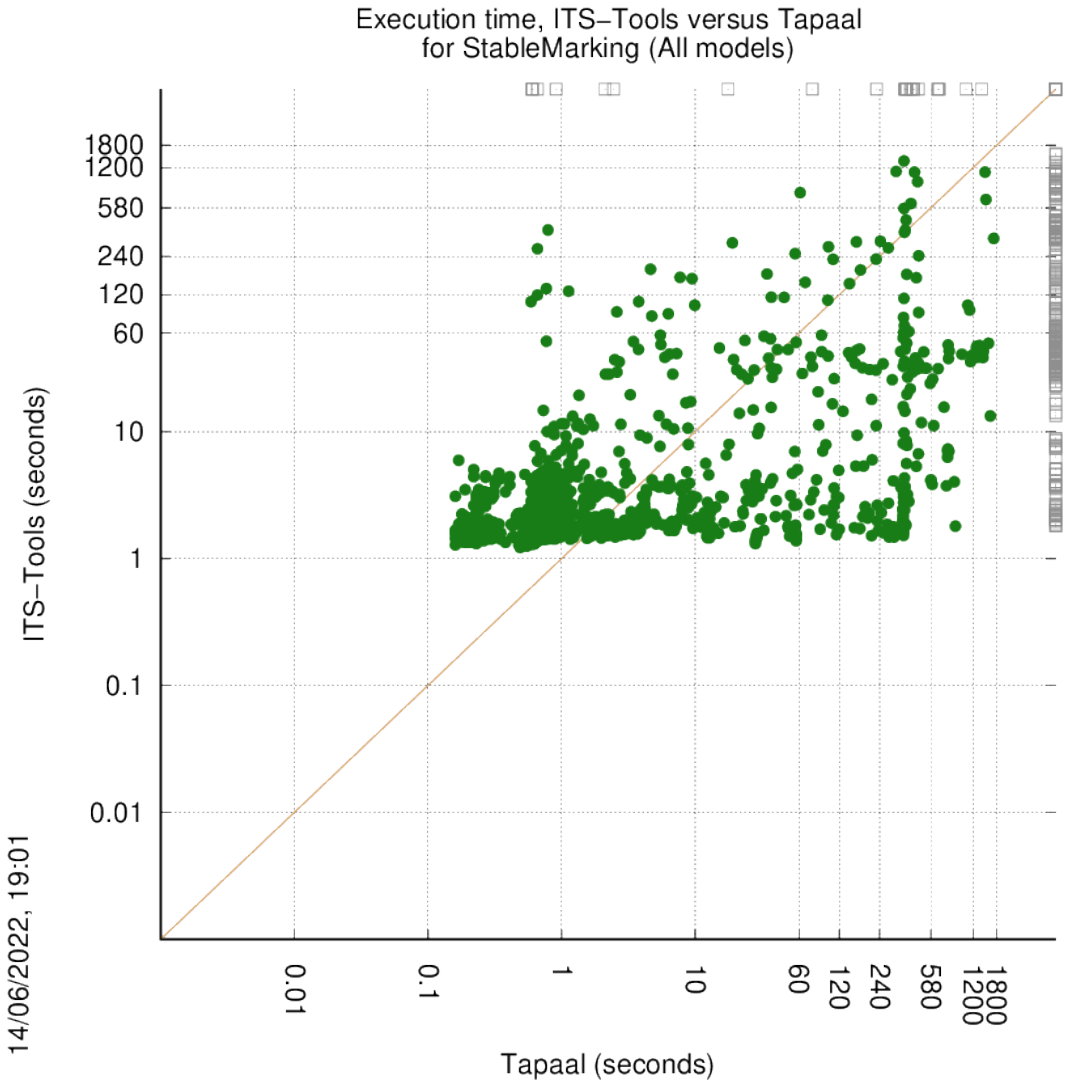

Figure 5: Results from MCC'2022 comparing ITS-Tools to Tapaal in time taken to solve the Stable Marking problems.

pendent petri-net editor final report., 2003. URL: https://pipe2.sourceforge.net/documents/PIPE-Report.pdf.

[9] B. Berthomieu, F. Vernadat, Time petri nets analysis with TINA, in: QEST, IEEE Computer Society, 2006, pp. 123–124.

[10] 4ti2 team, 4ti2—a software package for algebraic, geometric and combinatorial problems on linear spaces, 2023. URL: https://4ti2.github.io.

[11] Y. Thierry-Mieg, Symbolic and structural model-checking, Fundam. Informaticae 183 (2021) 319–342.

[12] L. M. de Moura, N. Bjørner, Z3: an efficient SMT solver, in: TACAS, volume 4963 of

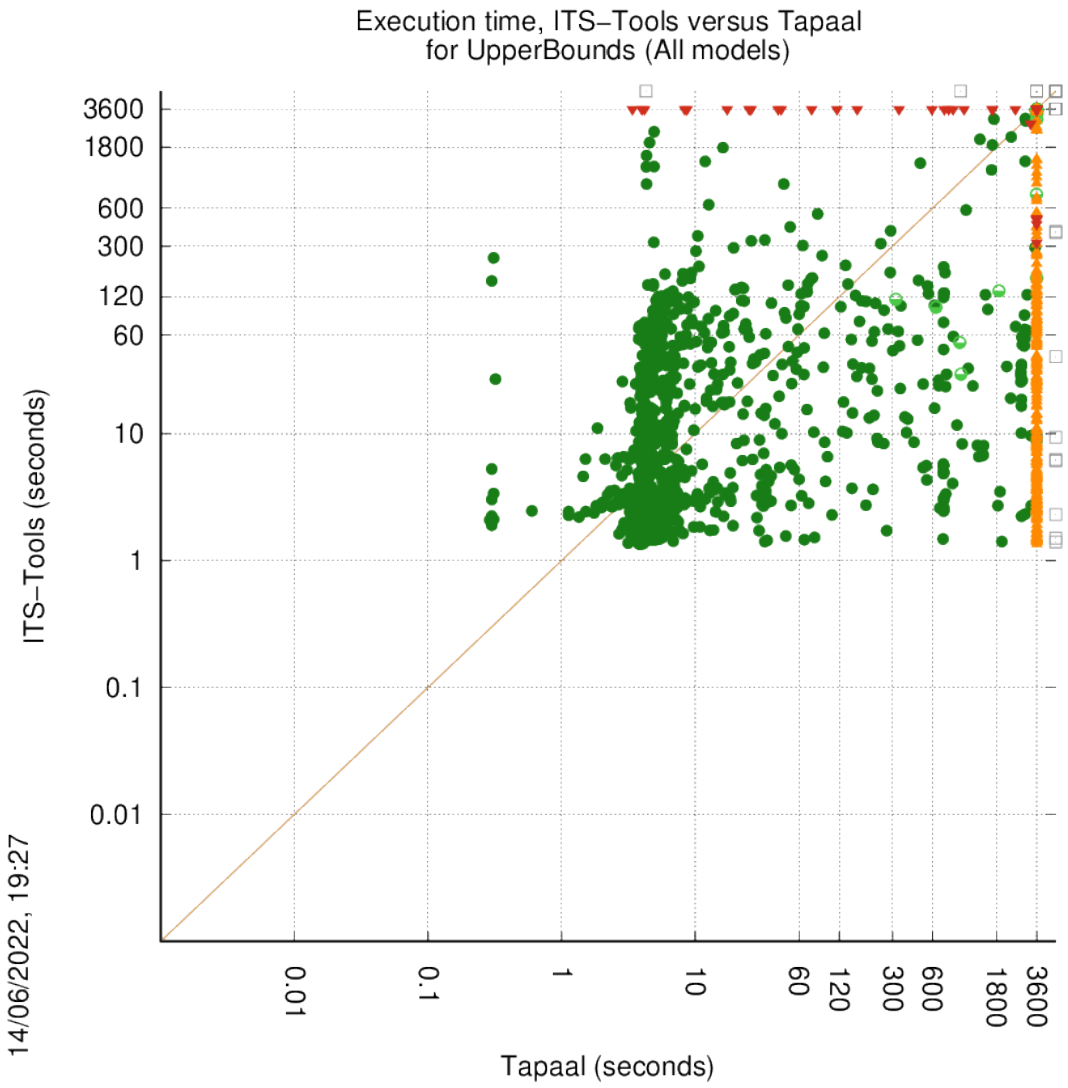

Figure 6: Results from MCC'2022 comparing ITS-Tools to Tapaal in time taken to solve the Upper Bounds problems.

Lecture Notes in Computer Science, Springer, 2008, pp. 337–340.

[13] A. Laarman, Stubborn transaction reduction, in: NFM, volume 10811 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 280–298.

[14] E. G. Amparore, S. Donatelli, G. Ciardo, Variable order metrics for decision diagrams in system verification, Int. J. Softw. Tools Technol. Transf. 22 (2020) 541–562.

[15] H. Garavel, Nested-unit petri nets, J. Log. Algebraic Methods Program. 104 (2019) 60–85.