Efficient Parameter Synthesis Using Optimized State Exploration Strategies

Étienne André, Hoang Gia Nguyen, and Laure Petrucci
Université Paris 13, LIPN, CNRS, UMR 7030, F-93430, Villetaneuse, France

***Abstract*—Parametric timed automata are a powerful formalism to reason about, model and verify real-time systems in which some constraints are unknown, or subject to uncertainty. Parameter synthesis using parametric timed automata is very sensitive to the state space explosion problem. To mitigate this problem, we propose two new exploration orders, i. e., the "ranking strategy" and the "priority based strategy", and compare them with existing strategies. We consider both complete parameter synthesis, and counterexample synthesis where the analysis stops as soon as some parameter valuations are found. Experimental results using IMITATOR show that our new strategies significantly outperform existing approaches, especially in the counterexample synthesis.**

***Index Terms*—parametric timed automata, exploration order, parametric model checking, IMITATOR**

1. Introduction

Real-time systems are notoriously difficult to design due to the complicated use of timing constraints, and must therefore be verified, e. g., using model checking. Model checking is a common approach to formally verify that a system which is described by a model, satisfies some property, described using formalisms such as properties expressed using, e. g., temporal logics.

Timed automata (TAs) [1] are a widely formalism used to model and verify real-time systems. TAs were successful in verifying models of complex distributed systems using power model checkers. TAs also have some limits when verifying systems only partially specified (typically when the timing constants are not yet known at an early design stage) or when timing constants are known with a limited precision only (although the robust semantics can help tackling some problems, see e. g., [2]). Parametric timed automata (PTAs) [3] leverage these drawbacks by allowing the use of timing parameters, hence allowing for modeling constants unknown or known with some imprecision.

By using parameters in the model, the model checking problem with a binary answer ("yes/no") becomes the *parameter synthesis* problem with a richer answer: a set of valuations for which a property holds. Parameter synthesis algorithms usually rely on the *parametric zone graph* (a parametric extension of the zone graph of TAs [4]) where states are pairs consisting of a discrete location and a parametric zone describing the set of possible parameter and clock valuations in this state (see e. g., [5], [6], [7]). The parametric zone graph is not only subject to the well-known state space explosion problem, but is usually even infinite. Indeed, most problems for PTAs are undecidable [8], including the emptiness of the parameter valuation set for which a given location is reachable ("EF-emptiness problem") [3]. A popular subclass of PTAs (called L/U-PTAs) was proposed in [5] with additional results in [7], [9], [10] but, despite decidability of the EF-emptiness problem, exact synthesis is intractable in practice [7], and the parametric zone graph is infinite for this subclass too.

Depth-first search (DFS) and breadth-first search (BFS) are popular exploration orders of model checking algorithms. By observation in practice, many authors (e. g., [11], [12]) showed that using BFS is much more efficient than DFS for checking reachability properties in TAs.

In [13], the authors show that in some cases, BFS can explore an exponential number of unnecessary states in TAs: this happens when a zone is found after another state with a strictly smaller zone (and the same discrete location) is found. We call this state with smaller zone a *redundant state* and this phenomenon an *inefficient phenomenon*.

Contribution. We study here various exploration orders to address efficient parameter synthesis for PTAs. By taking a different exploration to reach the larger zone first, we propose two main exploration strategies to reduce the inefficient phenomenon and increase the efficiency of parameter synthesis in PTAs.

The first exploration order we propose is a *parametric ranking strategy*, inspired by the ranking system of [13]. The second is a *parametric priority strategy*, which explores the biggest zone first in order to avoid the inefficient phenomenon and then stop the exploration from a small zone by correcting the inefficient phenomenon. Note that, in the worst case, these strategies explore unnecessary visited parametric zones exponentially. We also compare these exploration orders with the classical BFS strategy. We perform extensive experiments using the IMITATOR software [14] that takes as input parametric timed automata. First, we show that our new strategies always outperform the BFS strategy. Second, when using an additional existing state space optimization called "convex state merging" [15] (that can be used only for reachability properties), BFS becomes best again. However, for counter-example synthesis (i. e., try to find *some* parameter valuations instead of all), our

This work is partially supported by the ANR national research program PACS (ANR-14-CE28-0002).

exploration strategies significantly outperform BFS, with an average speed-up of 5.

Related works. As noted in [13], the exploration order problem was addressed in the context of state-caching focusing on limiting the number of stored nodes at nodes exploring cost [16], [17], [18], and state-space fragmentation [11], [12], [19], [20]. In [21], a value has been added to guide the exploration in priced timed automata, which has been reused in [13], and that we reuse in our second exploration strategy. Also note that the exploration order was considered in several works in the framework of *distributed* model checking for TAs [11], [12], [20], where it seems that BFS is the most optimal exploration order. Zone inclusion was also considered in [22] in multi-core model checking of TAs. To the best of our knowledge, comparing exploration strategies was never considered for PTAs or more generally for parametric timed formalisms. While we partially rely on exploration strategies for TAs, the differences of data structures (DBMs [4] cannot be used in PTAs) and the specificities of the symbolic zones for PTAs (that include not only clock valuations but also parameter valuations) make it important to study these strategies for PTAs.

Outline. We first recall the necessary definitions and the parametric zone inclusion algorithm for parametric timed automata in Section 2. Then, we introduce in Sections 3 and 4, parametric ranking strategy and parametric priority strategy respectively to limit the inefficient phenomenon during exploration. Section 5 provides experimental results of our approaches. Finally, we conclude in Section 6.

2. Preliminaries

2.1. Parameter Constraints

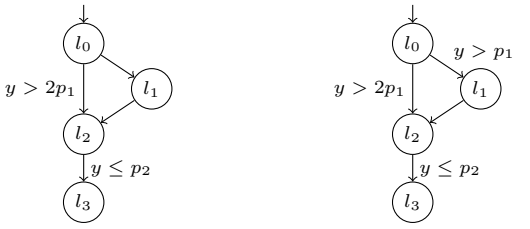
We assume a set $X = \{x_1, \ldots, x_H\}$ of *clocks*, i. e., real-valued variables that evolve at the same rate. A clock valuation w is a function $w : X \to \mathbb{R}_+$. We denote by $X = 0$ the conjunction of equalities that assigns 0 to all clocks in X.

We assume a set $P = \{p_1, \ldots, p_M\}$ of *parameters*, i. e., unknown constants. A *parameter valuation* v is a function $v : P \to \mathbb{Q}_+$. We will often identify a valuation v with the *point* $(v(p_1), \ldots, v(p_M))$.

An *inequality* over X and P is $e \bowtie 0$, where $\bowtie \in \{<, \leq, \geq, >\}$, and e is a linear term $\sum_{1 \leq i \leq N} \alpha_i z_i + d$ for some $N \in \mathbb{N}$, where $z_i \in X \cup P$, $\alpha_i \in \mathbb{Q}$, for $1 \leq i \leq N$, and $d \in \mathbb{Q}$. A (linear) constraint over X and P is a set of inequalities over X and P. We define in a similar manner inequalities and constraints over P. A *guard* is a set of inequalities each of them referring to exactly one clock.

Given a parameter valuation v, $C[v]$ denotes the constraint over X obtained by replacing each parameter p in C with $v(p)$. Likewise, given a clock valuation w, $C[v][w]$ denotes the expression obtained by replacing each clock x in $C[v]$ with $w(x)$. Given two constraints C_1 and C_2, we write $C_2 \subseteq C_2$ whenever, for any v, w, $C_1[v][w]$ evaluates to true implies $C_2[v][w]$ evaluates to true. *True* denotes the constraint made of all clock and parameter valuations.

We define the *time elapsing* of C, denoted by $C^{\nearrow}$, as the constraint over X and P obtained from C by delaying an arbitrary amount of time. Given $R \subseteq X$, we define the *reset* of C, denoted by $[C]_R$, as the constraint obtained from C by resetting the clocks in R, and keeping the other clocks unchanged.



(a) A PTA example (b) Modified version

Figure 1: Examples

Definition 1. A PTA $\mathcal{A}$ is a tuple $\mathcal{A} = (\Sigma, L, l_0, X, P, I, E)$, where: *i*) Σ is a finite set of actions, *ii*) L is a finite set of locations, *iii*) $l_0 \in L$ is the initial location, *iv*) X is a set of clocks, *v*) P is a set of parameters, *vi*) I is the invariant, assigning to every $l \in L$ a guard $I(l)$, and *vii*) E is a set of edges (l, g, a, R, l') where $l, l' \in L$ are the source and target locations, g is the transition guard, $a \in \Sigma$, and $R \subseteq X$ is a set of clocks to be reset.

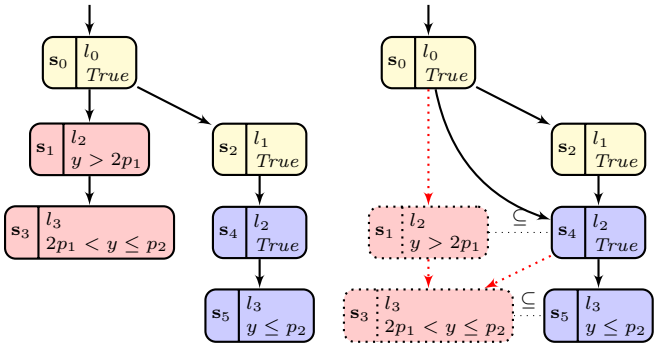
Example 1. Fig. 1a depicts an example of PTA with two clocks x and y (x does not appear on any guard nor invariant) and two parameters p_1 and p_2. Action labels are not shown. The transition from l_0 to l_2 is guarded by $y > 2p_1$ while the transition from l_2 to l_3 is guarded by $y \leq p_2$.

2.2. Symbolic Semantics

A symbolic state is a pair (l, C) with l a location, and C a constraint over $X \cup P$ or *zone*. The initial state of $\mathcal{A}$ is $\mathbf{s}_0 = (l_0, (X = 0)^{\nearrow} \wedge I(l_0))$, i. e., clocks are initially set to 0, and can evolve as long as $I(l_0)$ is satisfied. The computation of the state space is as follows: Given a symbolic state $\mathbf{s} = (l, C)$, $\mathsf{Succ}(\mathbf{s}) = \{(l', C') \mid \exists (l, g, a, R, l') \in E \text{ s.t. } C' = \left([(C \wedge g)]_R\right)^{\nearrow} \cap I(l')\}$.

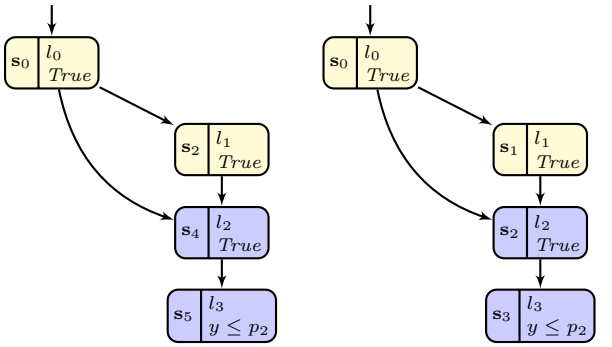
A symbolic run of a PTA is an alternating sequence of symbolic states and edges of the form $\mathbf{s}_0 \overset{e_0}{\Rightarrow} \mathbf{s}_1 \overset{e_1}{\Rightarrow} \cdots \overset{e_{m-1}}{\Rightarrow} \mathbf{s}_m$, such that for all $i = 0, \ldots, m-1$, $e_i \in E$, and $\mathbf{s}_i \overset{e_i}{\Rightarrow} \mathbf{s}_{i+1}$ is such that $\mathbf{s}_{i+1}$ belongs to $\mathsf{Succ}(\mathbf{s}_i)$ and is obtained via edge e_i. In the following, we simply refer to the symbolic states belonging to a run of $\mathcal{A}$ starting from $\mathbf{s}_0$ as states of $\mathcal{A}$. The *parametric zone graph* $\mathsf{PZG}(\mathcal{A})$ of a PTA $\mathcal{A}$ is made of the states of $\mathcal{A}$, and there is an edge in $\mathsf{PZG}(\mathcal{A})$ from $\mathbf{s}_i$ to $\mathbf{s}_j$ whenever $\mathbf{s}_j \in \mathsf{Succ}(\mathbf{s}_i)$.

In general, and in contrast to TAs [4], the parametric zone graph of PTAs is infinite, as most decision problems for PTAs, including the emptiness of the valuations set for



(a) Without parametric zone inclusion

(b) With parametric zone inclusion



(c) With parametric zone inclusion - result

(d) Ideal exploration order

Figure 2: Parametric zone graphs of Fig. 1a where the number n in a state label $\mathbf{s}_n$ reflects the exploration order

which a given location is reachable, are undecidable (see [8] for a survey).

2.3. Parametric Zone Inclusion Algorithm

Similar to timed automata's zone inclusion, *parametric zone inclusion* is an optimization technique relying on the parametric zone graph. That is, for some properties (including reachability and safety), given two reachable states $\mathbf{s}_1 = (l_1, C_1)$ and $\mathbf{s}_2 = (l_2, C_2)$, whenever $l_1 = l_2$ and $C_1 \subseteq C_2$, it is safe to replace $\mathbf{s}_1$ with $\mathbf{s}_2$ in the analysis. This inclusion check is even more costly in PTAs than its counterparts in TAs, but it is usually compensated by the performance improvement obtained by the decrease of the number of symbolic states to consider.

Algorithm 1 describes the standard state exploration algorithm with zone inclusion for PTA. It explores the infinite abstract parametric zone graph of PTA $\mathcal{A}$ from its initial location. The intuition of parametric zone inclusion is to stop the exploration of a small zone whenever a larger zone with the same location is explored. Therefore, in order to look up information of visited states having smaller zones at a certain location and do the zone inclusion, Algorithm 1 maintains a set of waiting states $\mathcal{W}$ and a graph $\mathcal{G}$ containing both visited states and transitions between them. In Algorithm 1, two situations can lead to zone inclusion:

The first is the classical one, at line 14: if a large zone has already been explored earlier, it subsumes the smaller zone being explored, which will be included by the larger zone with same location. Thus, only a transition from (l, C) to the larger zone (l', C_{Larger}) is added to $\mathcal{G}$, and not the newly computed state with a smaller zone (l', C').

The second situation where the inefficient phenomenon happens is when a larger zone is explored *after* exploring smaller zones. At line 9, the algorithm looks for the previous smaller parametric zones in the set of visited states $\mathcal{G}$, then removes states with smaller zones with its incoming and outgoing transitions. Also, transitions from parents of the smaller zones to the bigger zone and from the bigger zone to the children of the smaller zones are added at lines 10 and 11.

Note that, by exploring the bigger zones, the smaller zones and their successors or subtrees will eventually be pruned. Within the second situation, the parametric zone inclusion algorithm stores fewer nodes (i. e., symbolic states of the parametric zone graph) but this overhead of smaller zone removal procedure slightly influences the performance of the parametric zone inclusion algorithm.

Experiments will be reported in Section 5, where the performances of the different inclusions will be compared.

Note that Algorithm 1 does not mention any exploration order for any specific property checking. Choosing the exploration order will affect the performance of the algorithm, inefficient phenomenon and number of nodes visited by the algorithm and stored in the sets $\mathcal{W}$ and $\mathcal{G}$.

Example 2. We reuse in Fig. 1a a part of a parameterized version of the FDDI case study of [13]. Let us consider two different exploration strategies. The first parametric zone graph in Fig. 2a is explored by the standard BFS exploration order and the other in Fig. 2c by BFS with the parametric zone inclusion. Here we can see that by using parametric zone inclusion, the number of states to be explored is often reduced. Let us explain how Algorithm 1 works on the example in Fig. 1a using Fig. 2c. The algorithm starts at state $\mathbf{s}_0 : (l_0, True)$, the location is l_0 and the parametric zone is $True$ (i. e., the set of all clock and parameter valuations). Assume that the transition to l_2 is taken first. The algorithm reaches states $\mathbf{s}_1 : (l_2, y > 2p_1)$ and $\mathbf{s}_2 : (l_1, True)$. Later on, the algorithm reaches states $\mathbf{s}_3 : (l_3, 2p_1 < y \leq p_2)$ and $\mathbf{s}_4 : (l_2, True)$. At that stage, it happens that the parametric zone in $\mathbf{s}_4 : (l_2, True)$ is larger than the parametric zone in $\mathbf{s}_1 : (l_2, y > 2p_1)$ which has been visited previously. This previous exploration turns out to be useless, hence state $\mathbf{s}_1$ is removed and a transition from $\mathbf{s}_0$ to $\mathbf{s}_4$ is added. Finally, the algorithm does the same with states $\mathbf{s}_5$ and $\mathbf{s}_3$.

The ideal exploration is depicted in Fig. 2d. If the algorithm takes first the transition to location l_1 and then to l_2, the result is optimal. The goal of the remaining of this paper will be to get as close as possible to this optimal exploration order so as to avoid *redundant states*.

Algorithm 1: State exploration with parametric zone inclusion

Input: PTA $\mathcal{A} = (\Sigma, L, l_0, X, P, I, E)$
Output: parametric zone graph $\mathcal{Z}$ associated with the PTA $\mathcal{A}$

```
1  W ← {(l_0, C_0)}
2  G ← {(l_0, C_0)}
3  while W ≠ ∅ do
4      pick a state (l, C) from W
5      foreach outgoing state (l', C') from (l, C) do
6          if there is no (l', C_Larger) ∈ G such that C' ⊆ C_Larger then
7              add (l', C') to W and G
8              add transition (l, C) → (l', C') to G
9              foreach (l', C_Smaller) ∈ G such that C_Smaller ⊆ C' do
10                 add transitions to G: parent states of (l', C_Smaller) → (l', C') and
11                                       (l', C') → children states of (l', C_Smaller)
12             remove (l', C_Smaller) from W and G
13         else
14             foreach (l', C_Larger) ∈ G such that C' ⊆ C_Larger do add transition (l, C) → (l', C_Larger) to G
15 return G
```

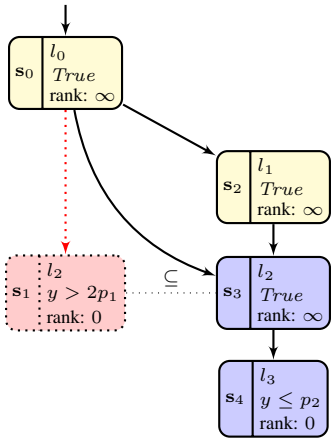


Figure 3: PZG with parametric ranking strategy

3. Parametric Ranking Strategy

In this section, we propose a novel exploration strategy for PTAs, inspired by the "ranking system" strategy that proved efficient for reducing the inefficient phenomenon in TAs [13]. As in [13], our parametric ranking strategy uses a priority value for each state. Then the algorithm explores the state with highest priority first. In case the inefficient phenomenon happens, the larger zone is assigned a higher priority than the smaller parametric zone and its previously explored subtrees.

The parametric ranking strategy in Algorithm 2 is an extension of the parametric zone inclusion of Algorithm 1 (differences are highlighted). Each newly explored state starts with being ranked with infinity (if its constraint is $True$)[1] or zero (otherwise) by Algorithm 3.

At lines 9 and 10, in order to stop the exploration of small parametric zones and their subtrees, the rank of the larger parametric zone is set higher than the highest rank of the small parametric zone and those in its subtree. This procedure is described in Algorithm 4: at line 3 it traverses all visited descendants of $(l', C_{Smaller})$ to get their highest rank. Since the larger zone has a higher rank, it will be explored before the smaller ones and their subtrees.

Example 3. Let us apply the parametric ranking strategy of Algorithm 2 to the example in Fig. 1a. The resulting parametric zone graph is shown in Fig. 3. Starting at state $(l_0, True)$, the algorithm ranks it with ∞. Then, states $\mathbf{s}_1 : (l_2, y > 2p_1)$ and $\mathbf{s}_2 : (l_1, True)$ are explored and added in the waiting set $\mathcal{W}$ with rank 0 and ∞ respectively. Hence, $\mathbf{s}_2$ with rank ∞ is explored first and leads to state $\mathbf{s}_3 : (l_2, True)$. At that stage, the algorithm detects that the zone of $\mathbf{s}_1 = (l_2, y > 2p_1)$ is smaller than that of $\mathbf{s}_3 = (l_2, True)$. The rank of $\mathbf{s}_3$ is ∞. The predecessor of $\mathbf{s}_1$ (i. e., $\mathbf{s}_0$) is connected to $\mathbf{s}_3$ and $\mathbf{s}_1$ is deleted. Finally, $\mathbf{s}_4$ (the successor of $\mathbf{s}_3$) is added with rank 0.

4. Parametric priority strategy

In [13], the authors indicate that with the "ranking system", there is no improvement if there are no $True$ zones in a model, compared to using the BFS exploration order. The same holds for our "parametric ranking" strategy. Indeed, first assigning the highest and lowest priority to each state, and then looking for visited states in order to find

1. Different from TAs, the initial constraint in PTAs is often not $True$ but a constraint over $X \cup P$ that also contains parameter constraints (for example $p_1 \leq p_2$). We assume w.l.o.g. that $True$ denotes this initial constraint (for example $p_1 \leq p_2$).

Algorithm 2: Ranking by parametric zone size

Input: PTA $\mathcal{A} = (\Sigma, L, l_0, X, P, I, E)$
Output: parametric zone graph $\mathcal{Z}$ associated with the PTA $\mathcal{A}$

```
1  r_0 ← init_rank(l_0, C_0)
2  W ← {((l_0, C_0), r_0)}
3  G ← {(l_0, C_0)}
4  while W ≠ ∅ do
5      pick a state ((l, C), r) with highest rank r from W
6      foreach outgoing state (l', C') from (l, C) do
7          r' ← init_rank(l', C')
8          if there is no ((l', C_Larger), r_L) ∈ G such that C' ⊆ C_Larger then
9              foreach ((l', C_Smaller), r_S) ∈ G such that C_Smaller ⊆ C' do
10                 r' ← max(r', max_rank((l', C_Smaller), r_S) + 1)
11             add ((l', C'), r') to W and G
12             add transition ((l, C), r) → ((l', C'), r') to G
13             foreach ((l', C_Smaller), r_S) ∈ G such that C_Smaller ⊆ C' do
14                 add transitions to G: parent nodes of ((l', C_Smaller), r_S) → ((l', C'), r') and
15                                       ((l', C'), r') → children states of ((l', C_Smaller), r_S)
16                 remove ((l', C_Smaller), r_S) from W and G
17         else
18             foreach ((l', C_Larger), r_L) ∈ G such that C' ⊆ C_Larger do
19                 add transition ((l, C), r) → ((l', C_Larger), r_L) to G
20 return G (without rank values)
```

Algorithm 3: $init\_rank(l, C)$

```
1 if C = True then return ∞ else return 0
```

Algorithm 4: $max\_rank((l, C), r)$

Output: $rank$ value

```
1 rank ← r
2 if ((l, C), r) ∉ W then
3     foreach ((l, C), r) → ((l', C'), r') in G do
4         rank ← max(rank, max_rank((l', C'), r'))
5 return rank
```

the highest rank in the subtrees (in large model where the subtrees become big) might be not efficient. Consider the example in Fig. 1a, where the guard of the transition from l_0 to l_1 is modified as in Fig. 1b. Its parametric zone graph is given in Fig. 4a.

In Fig. 4a, the parametric ranking algorithm ranks states from $\mathbf{s}_0$ to $\mathbf{s}_3$ with 0 continuously. The inefficient phenomenon is encountered as the parametric zone of $\mathbf{s}_4$ is larger than the one of $\mathbf{s}_1$ with the same location l_2. In this case, $\mathbf{s}_3$ is explored unnecessarily, similar to using the BFS exploration order.

However, after reaching states $\mathbf{s}_1$ and $\mathbf{s}_2$, if the algorithm explores the largest parametric zone first, i. e., $\mathbf{s}_2$, then $\mathbf{s}_4$ is reached earlier.

Hence, to avoid this inefficient phenomenon, we introduce a new strategy that explores the largest zone first in the sorted waiting list $\mathcal{W}$. Furthermore, in order to avoid traversing big subtrees to find the highest rank, we explore the largest zones until there is no inefficient phenomenon anymore. To do so, we use a simple inserting mechanism.

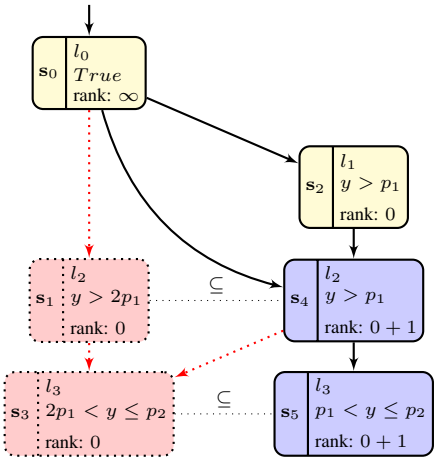
Before getting into the details of Algorithm 5, we explain the structure of the waiting list $\mathcal{W}$. First, $\mathcal{W}$ is ordered with decreasing zones. Hence there are two main parts in $\mathcal{W}$, the first (at the head) is the true zones part where all true zones are located. The other is the non-true zone part. Finally, since some non-true zones are incomparable, the true zones part can be seen as being itself composed of several parts each containing ordered comparable zones.

In Algorithm 5, the waiting list $\mathcal{W}$ described above, is sorted from largest to smallest zone by the inserting instructions from line 13 to line 17.

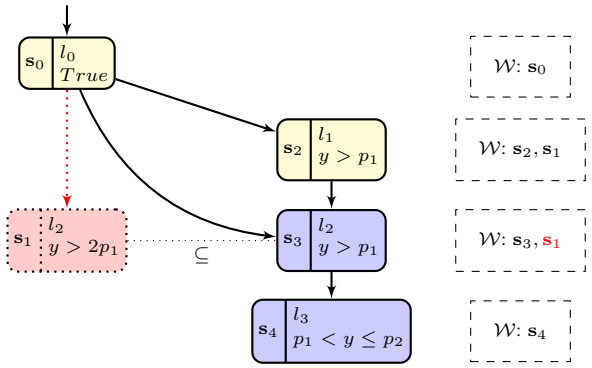
There are three possibilities. First, if C' is the true zone (which has the highest priority), (l', C') is inserted at the beginning of list $\mathcal{W}$. Next, if C' is not a true zone, (l', C') is inserted before the first smaller zone found in $\mathcal{W}$. Finally, in case all zones in $\mathcal{W}$ are incomparable with C', (l', C') is added at the end of list $\mathcal{W}$.

For better performances, the implementation uses an additional index (not described in Algorithm 5) storing information on the sets of comparable zones, for faster state insertion and comparison by avoiding repeated zone constraint computation.

Example 4. Let us apply Algorithm 5 to the example in



(a) Parametric ranking strategy



(b) Parametric priority strategy

Figure 4: Comparing our two strategies

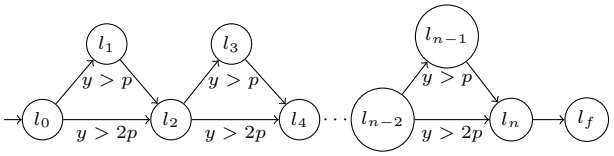
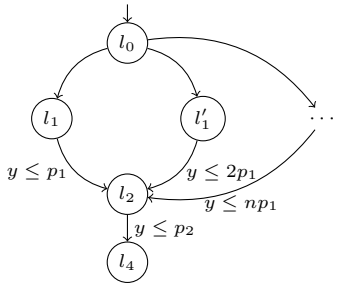


Figure 5: Blowup example

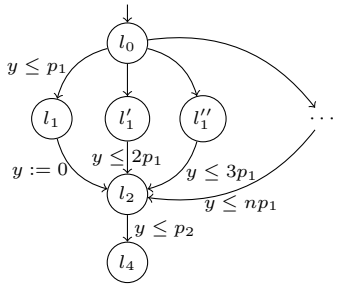
Fig. 1b. Fig. 4b shows both its parametric zone graph and the waiting list $\mathcal{W}$. The algorithm starts with state $\mathbf{s}_0$ in waiting list $\mathcal{W}$ and reaches the states $\mathbf{s}_1$ and $\mathbf{s}_2$ which are inserted into $\mathcal{W}$ decreasingly. Because the zone $(y > 2p_1)$ in $\mathbf{s}_1$ is smaller than the zone $(y > p_1)$ in $\mathbf{s}_2$ then $\mathbf{s}_2$ appears before $\mathbf{s}_1$ in $\mathcal{W}$. Then, the algorithm picks $\mathbf{s}_2$ from the head of list $\mathcal{W}$ and generates $\mathbf{s}_3$. It detects that $\mathbf{s}_3$ and $\mathbf{s}_1$ have the same location, and the parametric zone in $\mathbf{s}_3 : (l_2; y > p_1)$ is larger than the zone in $\mathbf{s}_1 : (l_2; y > 2p_1)$. Consequently, the state $\mathbf{s}_1$ is removed from $\mathcal{W}$ and $\mathcal{G}$ and the state $\mathbf{s}_3$ is inserted at the beginning of $\mathcal{W}$. Finally, the exploration of $\mathbf{s}_1$ is stopped and $\mathbf{s}_4$ is reached.

Example 5. Consider the inefficient phenomenon in Fig. 1b repeated n times as in Fig. 5 with parameter $p > 0$. Then it is inefficient for the ranking strategy, BFS. The performance of each algorithm with this model is given in Section 5.

However, our approaches still have some drawbacks. First, our algorithms base on the BFS so that before ranking, from a state, it has to reach all its descendants. This blind exploration might cause the inefficient phenomenon to happen, as illustrated in the previous example. Second, there might exist many paths between a pair of states that have equal parametric zones at start and different at the end, or some paths having small parametric zones in the beginning of the path that become larger after taking reset transitions as in Fig. 6a and Fig. 6b respectively. In Fig. 6a, the parametric zones of l_1, l'_1, l''_1 $\ldots l^n_1$ are equal. Thus, the algorithms explore from the path from l_1 to l^n_1, but at



(a)



(b)

Figure 6: Inefficiency in largest zone first like algorithms

the first state l_1, the parametric zone of l_2 reached has the smallest parametric zone. Hence, it causes the inefficient phenomenon repeatedly. In Fig. 6b, after all descendants of l_0 are reached, the parametric zone in l_1 is the smallest, then the path from l_1 is explored last. But it should be explored first, because the parametric zone at l_2 is reached from l_1 and is the biggest one due to resetting the clock y on the transition. Thus it causes the inefficient phenomenon.

Algorithm 5: Parametric priority strategy algorithm

Input: PTA $\mathcal{A} = (\Sigma, L, l_0, X, P, I, E)$
Output: parametric zone graph $\mathcal{Z}$ associated with the PTA $\mathcal{A}$

$\mathcal{W} \leftarrow \{(l_0, C_0)\}$
$\mathcal{G} \leftarrow \{(l_0, C_0)\}$
while $W \neq \emptyset$ **do**
  pick the first state (l, C) from $\mathcal{W}$
  foreach *outgoing state* (l', C') *from* (l, C) **do**
    if *there is no* $(l', C_{Larger}) \in \mathcal{G}$ *such that* $C' \subseteq C_{Larger}$ **then**
      add (l', C') to $\mathcal{G}$
      add transition $(l, C) \rightarrow (l', C')$ to $\mathcal{G}$
      foreach $(l', C_{Smaller}) \in \mathcal{G}$ *such that* $C_{Smaller} \subseteq C'$ **do**
        add transitions to $\mathcal{G}$: parent states of $(l', C_{Smaller}) \rightarrow (l', C')$ and
          $(l', C') \rightarrow$ children states of $(l', C_{Smaller})$
        remove $(l', C_{Smaller})$ from $\mathcal{W}$ and $\mathcal{G}$
      if $C' =$ `true` **then** insert (l', C') before the head of $\mathcal{W}$
      else if $C' \neq$ `true` **then**
        insert (l', C') before the first state (l_S, C_S) with a smaller zone $C_S \subseteq C'$ found in $\mathcal{W}$
      else
        insert (l', C') at the end of $\mathcal{W}$
    else
      foreach $(l', C_{Larger}) \in \mathcal{G}$ *such that* $C' \subseteq C_{Larger}$ **do** add transition $(l, C) \rightarrow (l', C_{Larger})$ to $\mathcal{G}$
return $\mathcal{G}$

5. Experimental Evaluation

To evaluate the performances of the proposed exploration orders experimentally, we compare them with one another, as well as with the standard BFS exploration strategy.

We implemented our algorithms in IMITATOR [14][2], and ran our experiments on an Intel core 2 duo P8600 at 2.4 GHz with 4 GiB of RAM. Polyhedra operations are performed using the PPL library [23].

Our benchmarks come from the IMITATOR benchmarks library and include hardware circuits (`AndOr`, `flipflop`, `spsmall`), network or software protocols (`BRP`, `FDDI-2`, `FDDI-4`, `Fischer-2`, `Fischer-3`, `F3`, `F4`, `Lynch-2`, `Lynch-5`, `critical-region`, `RCP`), real-time systems (`Thales-1`, `Thales-3`, `Sched2.`i`.`j), variants of a producer-consumer (`Pipeline` [24]), and the additional `blowup` example from Fig. 5 with 1001 locations.

We mainly focus on reachability synthesis, called the EF-synthesis problem: "find all parameter valuations for which a given location is reachable". A semi-algorithm was proposed in [3], [7], which we call EFsynth.

Additionally, we also focus on the counter-example synthesis: "find at least some parameter valuations for which a given location is reachable". Counter-example synthesis is of high practical importance, as it is often desirable to find at least some valuations for which a property holds (or is violated), not necessarily all of them. We implemented a procedure EFc-ex that stops as soon as some valuations are synthesized. Due to the undecidability of the EF-emptiness problem, neither EFsynth nor EFc-ex are guaranteed to terminate; note that they do for most of our experiments but not always. An advantage is that EFc-ex has a better termination than EFsynth (and in fact terminates in all our case studies) as a smaller part of the state space needs to be explored.

We will compare our new exploration strategies, i. e., parametric ranking strategy (RS) and parametric priority strategy (PRIOR), with the classical breadth-first search (BFS) strategy. In addition, we also consider a layer-based BFS strategy (LayerBFS), which is the historical strategy in IMITATOR, that computes all successor states of a given depth before computing all their successors at once. Although this is very close to the classical BFS strategy, some subtle implementation differences make its performances slightly different from BFS—and significantly when the merging heuristics (see Section 5.2 below) is used. Both BFS and LayerBFS come with two flavors: the bidirectional inclusion incl2 (which is as in Algorithm 1) and the mono-directional inclusion incl, where we only test whether the new state is included into an existing state, but not the other way round (i. e., lines 9–12 are discarded).

5.1. Comparison

We compare in Tables 1a and 1b our exploration strategies. From left to right in each table are model's name

2. Working version 2.9.2 (`explorder/5c40e39`). Sources, binaries, models, logs are available at www.imitator.fr/static/ICECCS17/.

followed by the computation times in seconds for each of the four strategies. Note that the green and yellow cells are the fastest and the second-fastest approaches respectively, and "T.O" stands for time-out after 15 minutes. The last line is the average using a normalized computation time for each benchmark (details are given in Appendix A).

From Table 1a, our two strategies RS and PRIOR behave almost the same for EFsynth, with a normalized average of 2.8. They both improve BFS by about 20 %, which shows the efficiency of our strategies.

From Table 1b, our strategies RS and PRIOR behave again almost the same for EFc-ex, but improve this time dramatically the computation time w.r.t. BFS, with a decrease of about 80 %. This shows the high efficiency of our strategies for counter-example synthesis.

A reason for the much better efficiency of our strategies for EFc-ex than EFsynth is that our strategies try to explore the largest zones first, and intuitively may lead much faster to a goal state. Then, once a goal state is found, EFc-ex stops and returns the associated parameter valuations, whereas EFsynth must explore the rest of the state space, for which the benefit of our strategies is milder.

5.2. Symbolic state merging

Our comparison is not entirely fair, as we did not use in our experiments another efficient optimization implemented in IMITATOR, i. e., state merging [15]. Given two states $\mathbf{s}_1 = (l_1, C_1)$ and $\mathbf{s}_2 = (l_2, C_2)$, it is possible to try to *merge* these states: $\mathbf{s}_1$ and $\mathbf{s}_2$ are *mergeable* if $l_1 = l_2$ and the polyhedron $C_1 \cup C_2$ is convex. The *merging* of $\mathbf{s}_1$ and $\mathbf{s}_2$ is then $(l_1, C_1 \cup C_2)$. In [15], we showed that merging states while computing the symbolic states keeps the soundness of the EFsynth algorithm; however, other algorithms usually lose their soundness when using state merging (this is the case of trace preservation synthesis, also called *inverse method* [15]). For example, parametric deadlock freeness checking [25] resembles EFsynth, but merging was not proved to be sound. Despite the very high cost of the mergeability test (up to 1000 times slower than other operations on polyehdra), state merging is often efficient because it can dramatically reduce the state space.

We compare in Tables 1c and 1d our exploration strategies with state merging. This time, our strategies are less efficient for exact synthesis using EFsynth (Table 1c); overall, the historical exploration strategy LayerBFS implemented in IMITATOR behaves about 2 times better than all other strategies. A reason comes from the cost of the merging: testing mergeability is very expensive, and LayerBFS iteratively tries to merge states once every state space depth ("layer") is completed, while other strategies try to merge states *for each newly computed symbolic state*, which is much more expensive. However, even when merging is used, our new strategies RS and PRIOR preserve a dramatic decrease of the computation time of more than 75 % for EFc-ex (Table 1d).

5.3. Final Interpretation

5.3.1. Exact synthesis. When one is interested in the exact synthesis (i. e., find all parameter valuations using EFsynth) for only reachability properties, then merging can be used, and the results are tabulated in Table 1c: the fastest strategy is clearly LayerBFS. Mono or bi-directional state inclusion do not fundamentally change the computation times, but in most cases (and in average), the bi-directional state inclusion is most efficient.

When one is interested in the exact synthesis for non-necessarily reachability properties, then merging cannot be used, and the results are tabulated in Table 1a: our two strategies RS and PRIOR perform best, 20 % faster than existing strategies.

5.3.2. Partial synthesis. When one is interested in finding some valuations only (i. e., EFc-ex), our two strategies RS and PRIOR perform significantly better than existing strategies, with a division of the computation time by 5 in average, when comparing with existing strategies.

Overall, PRIOR is 5.7 times faster than LayerBFS and 5.0 times faster than BFS using the normalized averages; for some case studies, the improvement w.r.t. BFS grows to 33 (`blowup`), 41 (`spsmall`), 72 (`Thales-3`), or even 522 (`pipeline-KP12-3-3`). Also note that, with the exception of `blowup`, the aforementioned three case studies are all industrial case studies.

6. Conclusion

In this paper, we have proposed two exploration order strategies to mitigate the inefficient phenomenon for the parameter synthesis problem: the parametric ranking strategy, and the parametric priority strategy. The intuition behind our strategies is to explore the large parametric zone first before reaching smaller zones.

Overall, our new strategies are reasonably faster than existing approaches for EFsynth, except when the merging heuristics is used (in which case BFS is more efficient). Our strategies become much faster than the literature for the counter-example synthesis using EFc-ex, up to 522 times faster for some industrial case studies. This suggests to use our new strategies as default for counter-example synthesis.

Future works. The waiting strategy of [13] could serve as a basis for future parametric strategies. In addition, mitigating the cost of merging states for strategies other than LayerBFS is on our agenda, by selecting the right time to perform this expensive test.

Furthermore, we would like to study exploration orders while taking advantage of recent multi-core technology, by adapting the non-parametric algorithm of [22].

References

[1] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[2] N. Markey, "Robustness in real-time systems," in *SIES*. IEEE Computer Society Press, 2011, pp. 28–34.

[3] R. Alur, T. A. Henzinger, and M. Y. Vardi, "Parametric real-time reasoning," in *STOC*. ACM, 1993, pp. 592–601.

[4] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," in *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, ser. LNCS, vol. 3098. Springer, 2003, pp. 87–124.

[5] T. Hune, J. Romijn, M. Stoelinga, and F. W. Vaandrager, "Linear parametric model checking of timed automata," *Journal of Logic and Algebraic Programming*, vol. 52-53, pp. 183–220, 2002.

[6] É. André and R. Soulat, *The Inverse Method*, ser. FOCUS Series in Computer Engineering and Information Technology. ISTE Ltd and John Wiley & Sons Inc., 2013, 176 pages.

[7] A. Jovanović, D. Lime, and O. H. Roux, "Integer parameter synthesis for timed automata," *TSE*, vol. 41, no. 5, pp. 445–461, 2015.

[8] É. André, "What's decidable about parametric timed automata?" in *FTSCS*, ser. CCIS, vol. 596. Springer, 2016, pp. 52–68.

[9] L. Bozzelli and S. La Torre, "Decision problems for lower/upper bound parametric timed automata," *Formal Methods in System Design*, vol. 35, no. 2, pp. 121–151, 2009.

[10] É. André and D. Lime, "Liveness in L/U-parametric timed automata," in *ACSD*. IEEE, 2017, pp. 9–18, to appear.

[11] G. Behrmann, T. Hune, and F. W. Vaandrager, "Distributing timed model checking – how the search order matters," in *CAV*, ser. LNCS, vol. 1855. Springer, 2000, pp. 216–231.

[12] G. Behrmann, "Distributed reachability analysis in timed automata," *STTT*, vol. 7, no. 1, pp. 19–30, 2005.

[13] F. Herbreteau and T. Tran, "Improving search order for reachability testing in timed automata," in *FORMATS*, ser. LNCS, vol. 9268. Springer, 2015, pp. 124–139.

[14] É. André, L. Fribourg, U. Kühne, and R. Soulat, "IMITATOR 2.5: A tool for analyzing robustness in scheduling problems," in *FM*, ser. LNCS, vol. 7436. Springer, 2012.

[15] É. André, L. Fribourg, and R. Soulat, "Merge and conquer: State merging in parametric timed automata," in *ATVA*, ser. LNCS, vol. 8172. Springer, 2013, pp. 381–396.

[16] P. Godefroid, G. J. Holzmann, and D. Pirottin, "State-space caching revisited," in *CAV*, ser. LNCS, vol. 663. Springer, 1992, pp. 178–191.

[17] G. Behrmann, K. G. Larsen, and R. Pelánek, "To store or not to store," in *CAV*, ser. LNCS, vol. 2725. Springer, 2003, pp. 433–445.

[18] S. Evangelista and L. M. Kristensen, "Search-order independent state caching," *ToPNoC*, vol. 4, pp. 21–41, 2010.

[19] C. Daws and S. Tripakis, "Model checking of real-time reachability properties using abstractions," in *TACAS*, ser. LNCS, vol. 1384. Springer, 1998, pp. 313–329.

[20] V. A. Braberman, A. Olivero, and F. Schapachnik, "ZEUS: A distributed timed model-checker based on KRONOS," *Electronic Notes in Theoretical Computer Science*, vol. 68, no. 4, pp. 503–522, 2002.

[21] G. Behrmann and A. Fehnker, "Efficient guiding towards cost-optimality in UPPAAL," in *TACAS*, ser. LNCS, vol. 2031. Springer, 2001, pp. 174–188.

[22] A. Laarman, M. C. Olesen, A. E. Dalsgaard, K. G. Larsen, and J. Van De Pol, "Multi-core emptiness checking of timed Büchi automata using inclusion abstraction," in *CAV*, ser. LNCS, vol. 8044. Springer, 2013, pp. 968–983.

[23] R. Bagnara, P. M. Hill, and E. Zaffanella, "The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems," *Science of Computer Programming*, vol. 72, no. 1–2, pp. 3–21, 2008.

[24] M. Knapik and W. Penczek, "Bounded model checking for parametric timed automata," *ToPNoC*, vol. 5, pp. 141–159, 2012.

[25] É. André, "Parametric deadlock-freeness checking timed automata," in *ICTAC*, ser. LNCS, vol. 9965. Springer, 2016, pp. 469–478.

Appendix

1. Normalizing the computation times

In order to compare all algorithms, we compute an average of the *normalized* computation times. However, due to the variety of the computation times (a same algorithm can use 0.014 s for a benchmark and 628 s for another one), performing the actual average wouldn't be fair: the behavior of the algorithms for the slowest benchmarks would have a much larger impact on the average than the fast benchmarks.

As a consequence, we normalize all computation times as follows: for each benchmark, we replace each computation time t by the division of this computation time t by the fastest algorithm for this benchmark i. e.,

$$\mathit{normalized} = \frac{t}{\min_{\mathit{algorithm}} t_{\mathit{algorithm}}}$$

That is, the fastest algorithm becomes 1 (which is the smallest possible value), and others timings give an idea of how slow they are w.r.t. the fastest. For example, a normalized value of 5.4 denotes that this algorithm is 5.4 times slower than the fastest algorithm for this benchmark.

In addition, to avoid that an algorithm gets a huge penalty for being, e. g., 200 times slower for one case study, we cap this normalized timing by 10. That is, the final formula becomes:

$$\mathit{normalized} = \min\left(\frac{t}{\min_{\mathit{algorithm}} t_{\mathit{algorithm}}}, 10\right)$$

Similarly, a timeout becomes 10 as well.

Finally, the average given in the tables is the average of all normalized times of an algorithm.

EFsynth (without merging)						
Benchmark Models	LayerBFS incl	LayerBFS incl2	BFS incl	BFS incl2	RS	PRIOR
AndOr	2.512	2.386	2.41	2.38	1.708	1.714
flipflop	121.108	121.026	102.42	109.412	139.822	140.193
BRP	377.913	322.67	370.74	304.277	174.038	160.079
Thales-1	30.802	37.75	44.114	37.593	41.575	40.476
Thales-3	627.956	T.O	759.987	T.O	636.823	597.57
Sched2.100.0	2.066	2.185	1.924	1.976	1.886	1.899
Sched2.100.2	148.169	93.138	T.O	90.158	249.373	259.895
Sched2.50.0	1.649	1.779	1.6	1.602	1.57	1.607
Sched2.50.2	28.137	27.119	217.399	23.344	36.81	35.26
FDDI-2	0.014	0.018	0.009	0.009	0.009	0.01
FDDI-4	1.315	1.252	1.1	1.092	1.455	1.285
Fischer-2	0.052	0.048	0.041	0.04	0.04	0.041
Fischer-3	0.521	0.538	0.48	0.497	1.172	1.316
Lynch-2	0.047	0.04	0.03	0.029	0.03	0.027
Lynch-5	7.359	7.429	7.817	7.346	8.859	7.867
F3	0.289	0.285	0.289	0.288	0.093	0.088
F4	21.813	22.573	37.558	20.626	108.629	96.983
Pipeline-KP12-2-3	21.975	31.642	18.516	29.735	19.489	19.07
Pipeline-KP12-2-5	T.O	T.O	T.O	T.O	T.O	T.O
Pipeline-KP12-3-3	T.O	T.O	T.O	T.O	T.O	T.O
RCP	1.105	1.147	1.099	1.088	0.093	0.095
spsmall	10.132	10.99	9.595	9.883	11.114	10.232
critical-region	T.O	T.O	T.O	T.O	T.O	T.O
critical-region-4	T.O	T.O	T.O	T.O	T.O	T.O
blowup	31.635	31.758	1.345	1.32	1.493	1.134
Average	3.47236	3.82884	3.7417	3.36366	2.85594	2.81208

(a) EFsynth (without merging)

EFsynth (with merging)						
Benchmark Models	LayerBFS incl	LayerBFS incl2	BFS incl	BFS incl2	RS	PRIOR
AndOr	1.635	1.618	1.734	1.758	5.365	5.305
flipflop	91.89	87.626	80.264	83.156	243.631	244.531
BRP	304.798	313.73	T.O	T.O	501.384	515.021
Thales-1	10.479	10.62	63.062	66.343	60.893	65.504
Thales-3	130.026	104.372	T.O	T.O	T.O	T.O
Sched2.100.0	2.55	2.771	7.996	8.302	17.352	17.327
Sched2.100.2	131.457	98.716	456.22	375.883	T.O	T.O
Sched2.50.0	2.078	2.213	6.185	6.405	14.45	14.231
Sched2.50.2	41.648	35.196	121.428	110.747	224.544	229.708
FDDI-2	0.015	0.01	0.009	0.008	0.011	0.008
FDDI-4	1.476	1.225	1.316	1.307	2.063	2.01
Fischer-2	0.048	0.046	0.046	0.039	0.042	0.042
Fischer-3	0.418	0.419	0.454	0.466	2.058	2.082
Lynch-2	0.031	0.036	0.028	0.026	0.044	0.037
Lynch-5	3.852	4.101	12.928	13.457	14.901	14.339
F3	0.244	0.245	0.409	0.412	0.143	0.137
F4	19.219	18.197	T.O	T.O	622.301	546.115
Pipeline-KP12-2-3	0.586	0.589	0.557	0.565	30.234	75.238
Pipeline-KP12-2-5	2.796	2.793	7.31	7.348	T.O	T.O
Pipeline-KP12-3-3	96.509	87.764	T.O	T.O	T.O	T.O
RCP	1.094	1.144	7.805	7.326	0.086	0.09
spsmall	1.893	1.591	2.623	2.547	12.498	13.677
critical-region	T.O	T.O	T.O	T.O	T.O	T.O
critical-region-4	T.O	T.O	T.O	T.O	T.O	T.O
blowup	313.649	33.018	345.481	1.365	1.709	1.343
Average	2.58396	2.52052	4.77691	4.38764	5.58949	5.59692

(b) EFc-ex (without merging)

EFc-ex (without merging)						
Benchmark Models	LayerBFS incl	LayerBFS incl2	BFS incl	BFS incl2	RS	PRIOR
AndOr	0.012	0.009	0.011	0.009	0.008	0.008
flipflop	0.061	0.064	0.059	0.055	0.029	0.028
BRP	2.874	2.476	2.944	2.863	0.198	0.188
Thales-1	4.854	4.753	6.189	5.748	0.126	0.119
Thales-3	16.638	16.397	19.968	20.247	0.237	0.232
Sched2.100.0	0.018	0.007	0.01	0.004	0.004	0.005
Sched2.100.2	0.008	0.007	0.004	0.004	0.005	0.004
Sched2.50.0	0.028	0.031	0.025	0.02	0.022	0.016
Sched2.50.2	0.028	0.036	0.023	0.024	0.016	0.015
FDDI-2	0.008	0.008	0.005	0.01	0.005	0.008
FDDI-4	0.377	0.329	0.291	0.287	0.091	0.078
Fischer-2	0.03	0.026	0.025	0.022	0.02	0.016
Fischer-3	0.097	0.097	0.097	0.092	0.057	0.059
Lynch-2	0.033	0.034	0.034	0.031	0.032	0.029
Lynch-5	7.408	7.619	7.912	7.329	8.847	7.829
F3	0.249	0.245	0.252	0.253	0.059	0.055
F4	4.086	3.786	6.543	4.16	0.364	0.311
Pipeline-KP12-2-3	0.313	0.344	0.245	0.27	0.031	0.025
Pipeline-KP12-2-5	3.825	4.83	2.988	4.526	0.049	0.037
Pipeline-KP12-3-3	21.927	33.184	18.229	31.204	0.042	0.042
RCP	0.51	0.506	0.454	0.453	0.024	0.02
spsmall	5.862	6.207	6.242	5.989	0.143	0.143
critical-region	0.148	0.121	0.081	0.073	0.016	0.018
critical-region-4	1.008	0.95	0.821	0.844	0.044	0.043
blowup	32.893	32.828	1.346	1.35	1.337	1.003
Average	6.03173	5.84164	5.28516	5.20335	1.13675	1.06026

(c) EFsynth (with merging)

EFc-ex (with merging)						
Benchmark Models	LayerBFS incl	LayerBFS incl2	BFS incl	BFS incl2	RS	PRIOR
AndOr	0.011	0.011	0.009	0.009	0.012	0.008
flipflop	0.074	0.065	0.061	0.060	0.031	0.028
BRP	1.906	2.215	3.950	4.112	0.198	0.197
Thales-1	3.191	3.182	8.461	9.068	0.126	0.123
Thales-3	10.826	11.233	55.746	58.321	0.244	0.234
Sched2.100.0	0.006	0.007	0.008	0.006	0.005	0.005
Sched2.100.2	0.009	0.01	0.006	0.006	0.004	0.01
Sched2.50.0	0.026	0.032	0.023	0.023	0.016	0.016
Sched2.50.2	0.036	0.037	0.024	0.024	0.017	0.017
FDDI-2	0.013	0.007	0.007	0.007	0.011	0.006
FDDI-4	0.364	0.365	0.35	0.349	0.088	0.08
Fischer-2	0.028	0.024	0.021	0.022	0.016	0.017
Fischer-3	0.085	0.093	0.084	0.09	0.072	0.069
Lynch-2	0.034	0.03	0.033	0.029	0.032	0.034
Lynch-5	3.893	4.182	13.038	13.533	14.778	14.344
F3	0.199	0.209	0.34	0.333	0.131	0.12
F4	3.048	3.151	45.059	53.607	0.948	0.867
Pipeline-KP12-2-3	0.096	0.086	0.075	0.082	0.029	0.028
Pipeline-KP12-2-5	0.278	0.267	0.247	0.273	0.038	0.04
Pipeline-KP12-3-3	0.639	0.639	0.825	0.841	0.043	0.041
RCP	0.532	0.575	3.117	3.154	0.022	0.024
spsmall	1.227	1.119	1.752	1.835	0.266	0.14
critical-region	0.111	0.112	0.082	0.072	0.025	0.015
critical-region-4	1.092	0.78	1.109	1.212	0.043	0.044
blowup	320.011	32.81	348.86	1.356	1.417	1.093
Average	5.05095	4.98319	5.21204	4.86171	1.29771	1.20662

(d) EFc-ex (with merging)

Table 1: Experiments