Revisiting Underapproximate Reachability for Multipushdown Systems

S. Akshay[1], Paul Gastin[2], Krishna S[1], and Sparsa Roychowdhury[1]

[1] IIT Bombay, Mumbai, India
{akshayss,krishnas,sparsa}@cse.iitb.ac.in
[2] ENS Paris-Saclay, France
paul.gastin@lsv.fr

Abstract. Boolean programs with multiple recursive threads can be captured as pushdown automata with multiple stacks. This model is Turing complete, and hence, one is often interested in analyzing a restricted class which still captures useful behaviors. In this paper, we propose a new class of bounded underapproximations for multipushdown systems, which subsumes most existing classes. We develop an efficient algorithm for solving the under-approximate reachability problem, which is based on efficient fix-point computations. We implement it in our tool BHIM and illustrate its applicability by generating a set of relevant benchmarks and examining its performance. As an additional takeaway BHIM solves the binary reachability problem in pushdown automata. To show the versatility of our approach, we then extend our algorithm to the timed setting and provide the first implementation that can handle timed multi-pushdown automata with closed guards.

Keywords: Multipushdown Systems, Underapproximate Reachability, Timed pushdown automata

1 Introduction

The reachability problem for pushdown systems with multiple stacks is known to be undecidable. However, multi-stack pushdown automata (MPDA hereafter) represent a theoretically concise and analytically useful model of multi-threaded recursive programs with shared memory. As a result, several previous works in the literature have proposed different under-approximate classes of behaviors of MPDA that can be analyzed effectively, such as *Round Bounded*, *Scope Bounded*, *Context Bounded* and *Phase Bounded* [1,2,3,4,5,6]. From a practical point of view, these underapproximations has led to efficient tools including, GetaFix [7], SPADE [8]. It has also been argued (e.g., see [9]) that such bounded underapproximations suffice to find several bugs in practice. In many such tools efficient fix-point techniques are used to speed-up computations.

We extend known fix-point based approaches by developing a new algorithm that can handle a larger class of bounded underapproximations than bounded context and bounded scope for multi-pushdown systems while remaining efficiently implementable. This algorithm works for a new class of underapproximate behaviors called *hole bounded* behaviors, which subsumes context or scope bounded underapproximations, and is orthogonal to phase bounded underapproximations. A "hole" is a maximal sequence of push operations of a fixed stack, interspersed with well-nested sequences of any stack. Thus, in a sequence $\alpha = \beta\gamma$ where $\beta = [push_1(push_2 push_3\ pop_3 pop_2)push_1(push_3 pop_3)]^{10}$ and $\gamma = push_2 push_1 pop_2 pop_1 (pop_1)^{20}$, β is a hole wrt stack 1. The suffix γ has 2 holes (the $push_2$ and the $push_1$). The number of context switches in α is > 50, and so is the number of changes in scope, while α is 3-hole bounded. A (k-)hole bounded sequence is one such, where, at any point of the computation, the number of holes are bounded (by k). We show that the class of hole bounded sequences subsumes most of the previously defined classes of underapproximations and is, in fact, contained in the very generic class of tree-width bounded sequences. This immediately shows decidability of reachability for our class.

Analyzing the more generic class of tree-width bounded sequences is often much more difficult; for instance, building bottom-up tree automata for this purpose does not scale very well as it explores a large (and often useless) state space. Our technique is radically different from using tree automata. Under the hole-bounded assumption, we pre-compute information regarding well-nested sequences and holes using fix-point computations and use them in our algorithm. Using efficient data structures to implement this approach, we develop a tool (BHIM) for Bounded Hole reachability in Multistack pushdown systems.

Highlights of BHIM.

• Two significant aspects of the fix-point approach in BHIM are: we efficiently solve the binary reachability problem for pushdown automata. i.e., BHIM computes all pairs of states (s, t) such that t is reachable from s with empty stacks. This allows us to go beyond reachability and handle some liveness questions; (ii) we pre-compute the set of pairs of states that are endpoints of holes. This allows us to greatly limit the search for an accepting run.

• While the fix-point approach solves (binary) reachability efficiently, it does not a priori produce a witness of reachability. We remedy this situation by proposing a backtracking algorithm, which cleverly uses the computations done in the fix-point algorithm, to generate a witness efficiently.

• BHIM is parametrized w.r.t the hole bound: if non-emptiness can be checked or witnessed by a well-nested sequence (this is an easy witness and BHIM looks for easy witnesses first, then gradually increases complexity, if no easy witness is found), then it is sufficient to have the hole bound 0; increasing this complexity measure as required to certify non-emptiness gives an efficient implementation, in the sense that we search for harder witnesses only when no easier witnesses (w.r.t this complexity measure) exist. In all examples as described in the experimental section, a small (less than 4) bound suffices and we expect this to be the case for most practical examples.

• Finally, extend our approach to handle timed multi-stack pushdown systems. This shows the versatility of our approach and also requires us to solve several technical challenges which are specific to the timed setting. Implementing this approach in BHIM makes it, to the best of our knowledge, the first tool that can analyze timed multi-stack pushdown automata (TMPDA) with closed guards.

We analyze the performance of BHIM in practice, by considering benchmarks from the literature, and generating timed variants of some of them. We modeled two variants of the Bluetooth example [10,8] and BHIM was able to detect three errors (of which it seems only two were already known). Likewise, for an example of a multiple producer consumer model, BHIM could detect bugs by finding witnesses having just 3 holes, while, it is unlikely that existing tools working on scope/context bounded underapproximations can handle them as the no. of switches in scope/context required would exceed 40 to find the bug. In the timed setting, one of the main challenges faced has been the unavailability of timed benchmarks; even in the untimed setting, many benchmarks were unavailable due to their proprietary nature. Nevertheless we tested our tool on 5 other benchmarks and 3 timed variants whose details, along with their parametric dependence plots, are given in Supplementary Material [11]. Due to lack of space proofs and technical details, especially in the timed setting are also in [11].

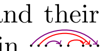
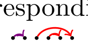
Related Work. Among other under-approximations, scope bounded [3] subsumes context and round bounded underapproximations, and it also paves path for GetaFix [7], a tool to analyze recursive (and multi-threaded) boolean programs. As mentioned earlier hole-boundedness strictly subsumes scope boundedness. On the other hand, GetaFix uses symbolic approaches via BDDs, which is orthogonal to the improvements made in this paper. Indeed, our next step would be to build a symbolic version of BHIM which extends the hole-bounded approach to work with symbolic methods. Given that BHIM can already handle synthetic examples with 12-13 holes (see [11]), we expect this to lead to even more drastic improvements and applicability. For sequential programs, a summary-based algorithm is used in [7]; summaries are like our well-nested sequences, except that well-nested sequences admit contexts from different stacks unlike summaries. As a result, our class of bounded hole behaviors generalizes summaries. Many other different theoretical results like phase bounded [1], order bounded [12] which gives interesting underapproximations of MPDA, are subsumed in tree-width bounded behaviors, but they do not seem to have practical implementations. Adding real-time information to pushdown automata by using clocks or timed stacks has been considered, both in the discrete and dense-timed settings. Recently, there has been a flurry of theoretical results in the topic [13,14,15,16,17]. However, to the best of our knowledge none of these algorithms have been successfully implemented (except [17] which implements a tree-automata based technique for single-stack timed systems) for multi-stack systems. One reason is that these algorithms do not employ scalable fix-point based techniques, but instead depend on region automaton-based search or tree automata-based search techniques.

2 Underapproximations in MPDA

A multi-stack pushdown automaton (MPDA) is a tuple $M = (\mathcal{S}, \Delta, s_0, \mathcal{S}_f, n, \Sigma, \Gamma)$ where, $\mathcal{S}$ is a finite non-empty set of locations, Δ is a finite set of transitions,

$s_0 \in \mathcal{S}$ is the initial location, $\mathcal{S}_f \subseteq \mathcal{S}$ is a set of final locations, $n \in \mathbb{N}$ is the number of stacks, Σ is a finite input alphabet, and Γ is a finite stack alphabet which contains $\perp$. A transition $t \in \Delta$ can be represented as a tuple (s, op, a, s'), where, $s, s' \in \mathcal{S}$ are respectively, the source and destination locations of the transition t, $a \in \Sigma$ is the label of the transition, and op is one of the following operations (1) nop, or no stack operation, (2) $(\downarrow_i \alpha)$ which pushes $\alpha \in \Gamma$ onto stack $i \in \{1, 2, \ldots, n\}$, (3) $(\uparrow_i \alpha)$ which pops stack i if the top of stack i is $\alpha \in \Gamma$.

For a transition $t = (s, \mathsf{op}, a, s')$ we write $\mathsf{src}(t) = s, \mathsf{tgt}(t) = s'$ and $\mathsf{op}(t) = \mathsf{op}$. At the moment we ignore the action label a but this will be useful later when we go beyond reachability to model checking. A *configuration* of the MPDA is a tuple $(s, \lambda_1, \lambda_2, \ldots, \lambda_n)$ such that, $s \in \mathcal{S}$ is the current location and $\lambda_i \in \Gamma^*$ represents the current content of i^{th} stack. The semantics of the MPDA is defined as follows: a run is accepting if it starts from the initial state and reaches a final state with all stacks empty. The language accepted by a MPDA is defined as the set of words generated by the accepting runs of the MPDA. Since the reachability problem for MPDA is Turing complete, we consider under-approximate reachability.

A sequence of transitions is called **complete** if each push in that sequence has a matching pop and vice versa. A **well-nested** sequence denoted ws is defined inductively as follows: a possibly empty sequence of nop-transitions is ws, and so is the sequence $t\ ws\ t'$ where $\mathsf{op}(t) = (\downarrow_i \alpha)$ and $\mathsf{op}(t') = (\downarrow_i \alpha)$ are a matching pair of push and pop operations of stack i. Finally the concatenation of two well-nested sequences is a well-nested sequence, i.e., they are closed under concatenation. The set of all well-nested sequences defined by an MPDA is denoted WS. If we visualize this by drawing edges between pushes and their corresponding pops, well-nested sequences have no crossing edges, as in  and , where we have two stacks, depicted with red and violet edges. We emphasize that a well-nested sequence can have well-nested edges from any stack. In a sequence σ, a push (pop) is called a **pending** push (pop) if its matching pop (push) is not in the same sequence σ.

Bounded Underapproximations. As mentioned in the introduction, different bounded under-approximations have been considered in the literature to get around the Turing completeness of MPDA. During a computation, a context is a sequence of transitions where only one stack or no stack is used. In *context bounded* computations the number of contexts are bounded [18]. A *round* is a sequence of (possibly empty) contexts for stacks $1, 2, \ldots, n$. *Round bounded* computations restrict the total number of rounds allowed [2,16,17]. *Scope bounded* computations generalize bounded context computations. Here, the context changes within any push and its corresponding pop is bounded [2,5,6]. A *phase* is a contiguous sequence of transitions in a computation, where we restrict pop to only one stack, but there are no restrictions on the pushes [1]. A phase bounded computation is one where the number of phase changes is bounded.

Tree-width. A generic way of looking at them is to consider classes which have a bound on the tree-width [19]. In fact, the notions of split-width/clique-width/tree-width of communicating finite state machines/timed push down systems has been explored in [20], [21]. The behaviors of the underlying system are then represented as graphs. It has been shown in these references that if the family of graphs arising from the behaviours of the underlying system (say S) have a bounded tree-width, then the reachability problem is decidable for S via, tree-automata. However, this does not immediately give rise to an efficient implementation. The tree-automata approach usually gives non-deterministic or bottom-up tree automata, which when implemented in practice (see [17]) tend to blow up in size and explore a large and useless space. Hence there is a need for efficient algorithms, which exist for more specific underapproximations such as context-bounded (leading to fix-point algorithms and their practical implementations [7]).

2.1 A new class of under-approximations

Our goal is to bridge the gap between having practically efficient algorithms and handling more expressive classes of under-approximations for reachability of multi-stack pushdown systems. To do so, we define a bounded approximation which is expressive enough to cover previously defined practically interesting classes (such as context bounded etc), while at the same time allowing efficient decidable reachability tests, as we will see in the next section.

Definition 1. *(Holes). Let σ be complete sequence of transitions, of length n in a* MPDA*, and let ws be a (possibly empty) well-nested sequence.*

- *A **hole** of stack i is a maximal factor of σ of the form $(\downarrow_i ws)^+$, where $ws \in \mathsf{WS}$. The maximality of the hole of stack i follows from the fact that any possible extension ceases to be a hole of stack i; that is, the only possible events following a maximal hole of stack i are a push $\downarrow_j$ of some stack $j \neq i$, or a pop of some stack $j \neq i$. In general, whenever we speak about a hole, the underlying stack is clear.*
- *A push $\downarrow_i$ in a hole (of stack i) is called a pending push at (i.e., just before) a position $x \leq n$, if its matching pop occurs in σ at a position $z > x$.*
- *A hole (of stack i) is said to be **open** at a position $x \leq n$, if there is a pending push $\downarrow_i$ of the hole at x. Let $\#_x(\mathsf{hole})$ denote the number of open holes at position x. The **hole bound** of σ is defined as $\max_{1 \leq x \leq |\sigma|} \#_x(\mathsf{hole})$.*
- *A* hole segment *of stack i is a prefix of a hole of stack i, ending in a ws, while an* atomic hole segment *of stack i is just the segment of the form $\downarrow_i ws$.*

As an example, consider the sequence σ in Figure 1 of transitions of a MPDA having stacks 1,2 (denoted respectively red and blue). We use superscripts for each push, pop of each stack to distinguish the ith push, jth pop and so on of each stack. There are two holes of stack 1 (red stack) denoted by the red patches,

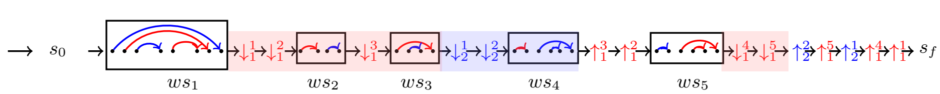


Fig. 1. A run σ with 2 holes (2 red patches) of the red stack and 1 hole (one blue patch) of the blue stack.

and one hole of stack 2 (blue stack) denoted by the blue patch. The subsequence $\downarrow_1^1\downarrow_1^2\ ws_2$ of the first hole is not a maximal factor, since it can be extended by $\downarrow_1^3\ ws_3$ in the run σ, extending the hole. Consider the position in σ marked with $\downarrow_2^1$. At this position, there is an open hole of the red stack (the first red patch), and there is an open hole of the blue stack (the blue patch). Likewise, at the position $\uparrow_1^5$, there are 2 open holes of the red stack (2 red patches) and one open hole of the blue stack 2 (the blue patch). The hole bound of σ is 3. The green patch consisting of $\uparrow_1^3$, $\uparrow_1^2$ and ws_5 is a pop-hole of stack 1. Likewise, the pops $\uparrow_2^2$, $\uparrow_1^5$, $\uparrow_2^1$ are all pop-holes (of length 1) of stacks 2,1,2 respectively.

Definition 2. (Hole Bounded Reachability Problem) *Given a* MPDA *and $K \in \mathbb{N}$, the K-hole bounded reachability problem is the following: Does there exist a K-hole bounded accepting run of the* MPDA*?*

Proposition 1. *The tree-width of K-hole bounded* MPDA *behaviors is at most $(2K+3)$.*

A detailed proof of this Proposition is given in Appendix A.1. Once we have this, from [19][16][17], decidability and complexity follow immediately. Thus,

Corollary 1. *The K-hole bounded reachability problem for* MPDA *is decidable in $\mathcal{O}(|\mathcal{M}|^{2K+3})$ where, $\mathcal{M}$ is the size of the underlying* MPDA.

Next, we turn to the expressiveness of this class wrt to the classical underapproximations of MPDA: first, the **hole** bounded class strictly subsumes **scope** bounded which already subsumes **context** bounded and **round** bounded classes. Also **hole** bounded MPDA and **phase** bounded MPDA are orthogonal.

Proposition 2. *Consider a* MPDA *M. For any K, let L_K denote a set of sequences accepted by M which have number of rounds or number of contexts or scope bounded by K. Then there exists $K' \leq K$ such that L_K is K' hole bounded. Moreover, there exist languages which are K hole bounded for some constant K, which are not K' round or context or scope bounded for any K'. Finally, there exists a language which is accepted by phase bounded* MPDA *but not accepted by hole bounded* MPDA *and vice versa.*

Proof. We first recall that if a language L is K-round, or K-context bounded, then it is also K'-scope bounded for some $K' \leq K$ [5,2]. Hence, we only show that scope bounded systems are subsumed by hole bounded systems.

Let L be a K-scope bounded language, and let M be a MPDA accepting L. Consider a run ρ of $w \in L$ in M. Assume that at any point i in the run ρ, $\#_i(\texttt{holes}) = k'$, and towards a contradiction, let, $k' > K$. Consider the leftmost open hole in ρ which has a pending push $\downarrow^p$ whose pop $\uparrow^p$ is to the right of i. Since $k' > K$ is the number of open holes at i, there are at least $k' > K$ context changes in between $\downarrow^p$ and $\uparrow^p$. This contradicts the K-scope bounded assumption, and hence $k' \leq K$.

To show the strict containment, consider the visibly pushdown language [22] given by $L^{bh} = \{a^n b^n (a^{p_1} c^{p_1+1} b^{p'_1} d^{p'_1+1} \cdots a^{p_n} c^{p_n+1} b^{p'_n} d^{p'_n+1}) \mid n, p_1, p'_1, \ldots, p_n, p'_n \in$

$\mathbb{N}\}$. A possible word $w \in L^{bh}$ is $a^3b^3\ a^2c^3b^2d^3\ a^2c^3bd^2\ ac^2bd^2$ with a, b representing push in stack 1,2 respectively and c, d representing the corresponding matching pop from stack 1,2. A run ρ accepting the word $w \in L^{bh}$ will start with a sequence of pushes of stack 1 followed by another sequence of pushes of stack 2. Note that, the number of the pushes n is same in both stacks. Then there is a group G consisting of a well-nested sequence of stack 1 (equal a and c) followed by a pop of the stack 1 (an extra c), another well-nested sequence of stack 2 (equal b and d) and a pop of the stack 2 (an extra d), repeated n times. From the definition of the `hole`, the total number of holes required in G is 0. But, we need 1 hole for the sequence of a's and another for the sequence of b's at the beginning of the run, which creates at most 2 holes during the run. Thus, the hole bound for any accepting run ρ is 2, and the language L^{bh} is 2-hole bounded.

However, L^{bh} is not k-scope bounded for any k. Indeed, for each $m \geq 1$, consider the word $w_m = a^m b^m (ac^2bd^2)^m \in L^{bh}$. It is easy to see that w_m is $2m$-scope bounded (the matching c, d of each a, b happens $2m$ context switches later) but not k-scope bounded for $k < 2m$. It can be seen that L^{bh} is not k-phase bounded either. Finally, $L' = \{(ab)^n c^n d^n \mid n \in \mathbb{N}\}$ with a, b and c, d respectively being push and pop of stack 1,2 is not hole-bounded but 2-phase bounded. $\square$

3 A Fix-point Algorithm for Hole Bounded Reachability

In the previous section, we showed that hole-bounded underapproximations are a decidable subclass for reachability, by showing that this class has a bounded tree-width. However, as explained in the introduction, this does not immediately give a fix-point based algorithm, which has been shown to be much more efficient for other more restricted sub-classes, e.g., context-bounded. In this section, we provide such a fix-point based algorithm for the hole-bounded class and explain its advantages. Later we discuss its versatility by showing extensions and evaluating its performance on a suite of benchmarks.

We describe the algorithm in two steps: first we give a simple fix-point based algorithm for the problem of 0-hole or *well-nested reachability*, i.e, reachability by a well-nested sequence without any holes. For the 0-hole case, our algorithm computes the *reachability relation*, also called the *binary reachability problem [23]*. That is, we accept all pairs of states (s, s') such that there is a well-nested run from s with empty stack to s' with empty stack. Subsequently, we combine this binary reachability for well-nested sequences with an efficient graph search to obtain an algorithm for K-hole bounded reachability.

Binary well-nested reachability for MPDA. Note that single stack PDA are a special case, since all runs are indeed well-nested.

1. **Transitive Closure**: Let $\mathcal{R}$ be the set of tuples of the form (s_i, s_j) representing that state s_j is reachable from state s_i via a nop discrete transition. Such a sequence from s_i to s_j is trivially *well-nested*. We take the TransitiveClosure of $\mathcal{R}$ using Floyd-Warshall algorithm [24]. The resulting set $\mathcal{R}_c$ of tuples answers the binary reachability for finite state automata (no stacks).

Algorithm 1: Algorithm for Emptiness Checking of hole bounded MPDA

Function $\texttt{IsEmpty}(M = (\mathcal{S}, \Delta, s_0, \mathcal{S}_f,\ n, \Sigma, \Gamma), K)$:
    Result: True or False
    $\texttt{WR} := \texttt{WellNestedReach}(M)$; \\Solves binary reachability for pushdown system
    if *some* $(s_0, s_1) \in \texttt{WR}$ *with* $s_1 \in \mathcal{S}_f$ **then**
        return *False*;
    forall $i \in [n]$ **do**
        $AHS_i := \emptyset$; $Set_i := \emptyset$;
        forall $(s, \downarrow_i(\alpha), a, s_1) \in \Delta$ *and* $(s_1, s') \in \texttt{WR}$ **do**
            $AHS_i := AHS_i \cup \{(i, s, \alpha, s')\}$; $Set_i := Set_i \cup \{(s, s')\}$;
        $HS_i := \{(i, s, s') \mid (s, s') \in \texttt{TransitiveClosure}(Set_i)\}$;
    $\mu := [s_0]$; $\mu.\texttt{NumberOfHoles} := 0$;
    $\texttt{SetOfLists}_{new} := \{\mu\}$; $\texttt{SetOfLists} := \emptyset$;
    do
        $\texttt{SetOfLists} := \texttt{SetOfLists} \cup \texttt{SetOfLists}_{new}$;
        $\texttt{SetOfLists}_{todo} := \texttt{SetOfLists}_{new}$; $\texttt{SetOfLists}_{new} := \emptyset$;
        forall $\mu' \in \textit{SetOfLists}_{todo}$ **do**
            if $\mu'.\texttt{NumberOfHoles} < K$ **then**
                forall $i \in [n]$ **do**
                    \\ Add hole for stack i
                    $\texttt{SetOfLists}_h := \texttt{AddHole}_i(\mu', HS_i) \setminus \texttt{SetOfLists}$;
                    $\texttt{SetOfLists}_{new} := \texttt{SetOfLists}_{new} \cup \texttt{SetOfLists}_h$;
            if $\mu'.\texttt{NumberOfHoles} > 0$ **then**
                forall $i \in [n]$ **do**
                    \\ Add pop for stack i
                    $\texttt{SetOfLists}_p := \texttt{AddPop}_i(\mu', M, AHS_i, HS_i, \texttt{WR}) \setminus \texttt{SetOfLists}$;
                    $\texttt{SetOfLists}_{new} := \texttt{SetOfLists}_{new} \cup \texttt{SetOfLists}_p$;
                    forall $\mu_3 \in \textit{SetOfLists}_p$ **do**
                        if $\mu_3.last \in \mathcal{S}_f$ *and* $\mu_3.\texttt{NumberOfHoles} = 0$ **then**
                            return *False*; \\If reached destination state
    while $\textit{SetOfLists}_{new} \neq \emptyset$;
    return *True*;

2. **Push-Pop Closure**: For stack operations, consider a push transition on some stack (say stack i) of symbol γ, enabled from a state s_1, reaching state s_2. If there is a matching pop transition from a state s_3 to s_4, which pops the same stack symbol γ from the stack i and if we have $(s_2, s_3) \in \mathcal{R}_c$, then we can add the tuple (s_1, s_4) to $\mathcal{R}_c$. The function WellNestedReach (Algorithm 2, Appendix B) repeats this process and the transitive closure described above until a fix-point is reached. Let us denote the resulting set of tuples by WR. Thus, we have

Lemma 1. $(s_1, s_2) \in \mathsf{WR}$ *iff* $\exists$ *a well-nested run in the* MPDA *from* s_1 *to* s_2.

Beyond well-nested reachability. A naive algorithm for K-hole bounded reachability for $K > 0$ is to start from the initial state s_0, and do a Breadth First Search (BFS), nondeterministically choosing between extending with a well-nested segment, creating hole segments (with a pending push) and closing hole segments (using pops). We accept when there are no open hole segments and reach a final state; this gives an exponential time algorithm. Given the exponential dependence on the hole-bound K (Corollary 1), this exponential blowup is unavoidable in the worst case, but we can do much better in practice. In particular, the naive algorithm makes arbitrary non-deterministic choices resulting in a blind exploration of the BFS tree.

In this section, we use the binary well-nested reachability algorithm as an efficient subroutine to limit the search in BFS to its reachable part (note that this is quite different from DFS as well since we do not just go down one path). The crux is that at any point, we create a new hole for stack i, *only* when (i) we know that we cannot reach the final state without creating this hole and (ii) we know that we can close all such holes which have been created. Checking (i) is easy, since we just use the WR relation for this. Checking (ii) blindly would correspond to doing a DFS; however, we precompute this information and simply look it up, resulting in a constant time operation after the precomputation.

Precomputing hole information. Recall that a *hole* of stack i is a maximal sequence of the form $(\downarrow_i \ ws)^+$, where ws is a well-nested sequence and $\downarrow_i$ represents a push of stack i . A *hole segment* of stack i is a prefix of a hole of stack i, ending in a ws, while an *atomic hole segment* of stack i is just the segment of the form $\downarrow_i ws$. A *hole-segment* of stack i which starts from state s in the MPDA and ends in state s', can be represented by the triple (i, s, s'), that we call a *hole triple*. We compute the set HS_i of all hole triples (i, s, s') such that starting at s, there is a hole segment of stack i which ends at state s', as detailed in lines (5-9) of Algorithm 1. In doing so, we also compute the set AHS_i of all atomic hole segments of stack i and store them as tuples of the form (i, s_p, α, s_q) such that s_p and s_q are the MPDA states respectively at the left and right end points of an atomic hole segment of stack i, and α is the symbol pushed on stack i ($s_p \xrightarrow{\downarrow_i(\alpha)ws} s_q$).

A guided BFS exploration. We start with a list $\mu_0 = [s_0]$ consisting of the initial state and construct a BFS exploration tree whose nodes are lists of bounded length. A list is a sequence of states and hole triples representing a K-hole bounded run in a concise form. If H_i represents a hole triple for stack i, then a list is a sequence of the form $[s, H_i, H_j, H_k, H_i, \ldots, H_\ell, s']$. The simplest kind of list is a single state s. For example, a list with 3 holes of stacks i, j, k is $\mu = [s_0, (i, s, s'), (j, r, r'), (k, t, t'), t'']$. The hole triples (in red) denote open holes in the list. The maximum number of open holes in a list is bounded, making the length of the list also bounded. Let $\mathsf{last}(\mu)$ represent the last element of the list μ. This is always a state. For a node v storing list μ in the BFS tree, if $v_1, \ldots v_k$ are its children, then the corresponding lists $\mu_1, \ldots \mu_k$ are obtained by extending the list μ by one of the following operations:

1. **Extend μ with a hole**. Assume there is a hole of some stack i, which starts at $\mathsf{last}(\mu) = s$, and ends at s'. If the list at the parent node v is $\mu = [\ldots, s]$, then for all $(i, s, s') \in HS_i$, we obtain the list $\mathsf{trunc}(\mu) \cdot \mathsf{append}[(i, s, s'), s']$ at the child node (i.e., we remove the last element s of μ, then append to this list the hole triple (i, s, s'), followed by s'). Algorithm 3 in Appendix describes this operation in more detail.
2. **Extend μ with a pop**. Suppose there is a transition $t = (s_k, \uparrow_i(\alpha), a, s'_k)$ from $\mathsf{last}(\mu) = s_k$, where μ is of the form $[s_0, \ldots, (h, u, v), (i, s, s'), (j, t, t') \ldots, s_k]$, such that there is no hole triple of stack i after (i, s, s'), we extend the run by matching this pop (with its push). However, to obtain the last pending push

of stack i corresponding to this hole, just HS_i information is not enough since we also need to match the stack content. Instead, we check if we can split the hole (i, s, s') into (1) a hole triple $(i, s, s_a) \in HS_i$, and (2) a tuple $(i, s_a, \alpha, s') \in AHS_i$. If both (1) and (2) are possible, then the pop transition t corresponds to the last pending push of the hole (i, s, s'). t indeed matches the pending push recorded in the atomic hole (i, s_a, α, s') in μ, enabling the firing of transition t from the state s_k, reaching s'_k. In this case, we add the child node with the list μ' obtained from μ as follows. We replace (i) s_k with s'_k, and (ii) (i, s, s') with (i, s, s_a), respectively signifying firing of the transition t and the "shrinking" of the hole, by shifting the end point of the hole segment to the left. When we obtain the hole triple (i, s, s) (the start and end points of the hole segment coincide), we may have uncovered the last pending push and thereby "closed" the hole segment completely. At this point, we may choose to remove (i, s, s) from the list, obtaining $[s_0, \dots, (h, u, v),\ (j, t, t') \dots, s'_k]$. For every such $\mu' = [s_0, \dots, (h, u, v), (i, s, s_a), (j, t, t'), \dots, s'_k]$ and all $(s'_k, s_m) \in WS$ we also extend μ' to $\mu'' = [s_0, \dots, (h, u, v), (i, s, s_a), (j, t, t'), \dots, s_m]$. Notice that the size of the list in the child node obtained on a pop, is either the same as the list in the parent, or is smaller. The details are in Algorithm 4.

The number of lists is bounded since the number of states and the length of the lists are bounded. The BFS exploration tree will thus terminate. Combining the above steps gives us Algorithm 1, whose correctness gives us:

Theorem 1. *Given a* MPDA *and a positive integer K, Algorithm 1 always terminates and answers "false" iff there exists a K-hole bounded accepting run of the* MPDA.

Complexity of the Algorithm. The maximum number of states of the system is $|\mathcal{S}|$. The time complexity of transitive closure is $\mathcal{O}(|\mathcal{S}|^3)$, using a Floyd-Warshall implementation. The time complexity of Algorithm 2, which uses the transitive closure, is $\mathcal{O}(|\mathcal{S}|^5) + \mathcal{O}(|\mathcal{S}|^2 \times (|\Delta| \times |\mathcal{S}|))$. To compute AHS for n stacks the time complexity is $\mathcal{O}(n \times |\Delta| \times |\mathcal{S}|^2)$ and to compute HS for n stacks the complexity is $\mathcal{O}(n \times |\mathcal{S}|^2)$. For multistack systems, each list keeps track of (i) the number of hole segments($\leq K$), and (ii) information pertaining to holes (start, end points of holes, and which stack the hole corresponds to). In the worst case, this will be $(2K+2)$ possible states in a list, as we are keeping the states at the start and end points of all the hole segments and a stack per hole. So, there are $\leq |\mathcal{S}|^{2K+3} \times n^{K+1}$ lists. In the worst case, when there is no K-hole bounded run, we may end up generating all possible lists for a given bound K on the hole segments. The time complexity is thus bounded above by $\mathcal{O}(|\mathcal{S}|^{2K+3} \times n^{K+1} + |\mathcal{S}|^5 + |\mathcal{S}|^3 \times |\Delta|)$.

Beyond Reachability. We can solve the usual safety questions in the (bounded-hole) underapproximate setting, by checking for underapproximate reachability on the product of the given system with the complement of the safe set. Given the way Algorithm 1 is designed, the fix-point algorithm allows us to go beyond reachability. In particular, we can solve several (increasingly difficult) variants of the repeated reachability problem, without much modification.

Consider the question : For a given state s and MPDA, does there exist a run ρ starting from s_0 which visits s infinitely often? This is decidable if we can decompose ρ into a finite prefix ρ_1 and an infinite suffix ρ_2 s.t. (1)Both ρ_1, ρ_2 are well-nested, or (2) ρ_1 is K-hole bounded complete (all stacks empty), and ρ_2 is well-nested, or (3) ρ_1 is K-hole bounded, and $\rho_2 = (\rho_3)^\omega$, where ρ_3 is K-hole bounded. It is easy to see that (1) is solved by two calls to WellNestedReach and choosing non-empty runs. (2) is solved by a call to Algorithm 1, modified so that we reach s, and then calling WellNestedReach. Lastly, to solve (3), first modify Algorithm 1 to check reachability to s with possibly non-empty stacks. Then run the modified algorithm twice : first start from s_0 and reach s; second start from s and reach s again.

4 Generating a Witness

We next focus on the question of generating a witness for an accepting run when our algorithm guarantees non-emptiness. This question is important to address from the point of view of applicability: if our goal is to see if bad states are reachable, i.e., non-emptiness corresponds to presence of a bug, the witness run gives the trace of how the bug came about and hence points to what can be done to fix it (e.g., designing a controller). We remark that this question is difficult in general. While there are naive algorithms which can explore for the witness (thus also solving reachability), these do not use fix-point techniques and hence are not efficient. On the other hand, since we use fix-point computations to speed up our reachability algorithm, finding a witness, i.e., an explicit run witnessing reachability, becomes non-trivial. Generation of a witness in the case of well-nested runs is simpler than the case when the run has holes, and requires us to "unroll" pairs $(s_0, s_f) \in \mathtt{WR}$ recursively and generate the sequence of transitions responsible for (s_0, s_f), as detailed in Algorithm 5.

Getting Witnesses from Holes. Now we move on to the more complicated case of behaviours having holes. Recall that in BFS exploration we start from the states reachable from s_0 by well-nested sequences, and explore subsequent states obtained either from (i) a hole creation, or (ii) a pop operation on a stack. Proceeding in this manner, if we reach a final configuration (say s_f), with all holes closed (which implies empty stacks), then we declare non-emptiness. To generate a witness, we start from the final state s_f reachable in the run (a leaf node in the BFS exploration tree) and *backtrack* on the BFS exploration tree till we reach the initial state s_0. This results in generating a witness run in the reverse, from the right to the left.

• Assume that the current node of the BFS tree was obtained using a pop operation. There are two possibilities to consider here (see below) depending on whether this pop operation closed or shrunk some hole. Recall that each hole has a left end point and a right end point and is of a specific stack i, depending on the pending pushes $\downarrow_i$ it has. So, if the MPDA has k stacks, then a list in the exploration tree can have k kinds of holes. The witness algorithm uses k stacks called *witness stacks* to correctly implement the backtracking procedure, to deal with k kinds of holes. Witness stacks should not be confused with the stacks of the MPDA.

• Assume that the current pop operation is closing a hole of kind i as in Figure 2. This hole consists of the atomic holes , and . The atomic hole consists of the push and the well-nested sequence (same for the other two atomic holes). Searching among possible push transitions, we identify the matching push associated with the current pop, resulting in closing the hole. On backtracking, this leads to a parent node with the atomic hole having as left end point, the push , and the right end point as the target of the ws . We push onto the witness stack i, a barrier (a delimiter symbol #) followed by the matching push transition and then the ws, . The barrier segregates the contents of the witness stack when we have two pop transitions of the same stack in the reverse run, closing/shrinking two different holes.

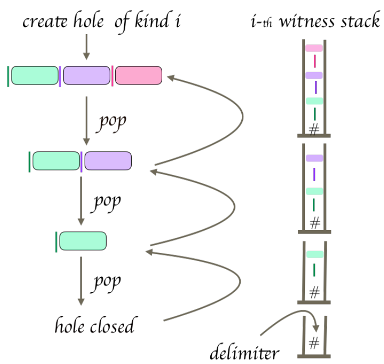


Fig. 2. Backtracking to spit out the hole in reverse. The transitions of the atomic hole are first written in the reverse order, followed by those of in reverse, and then of in reverse.

• Assume that the current pop operation is shrinking a hole of kind i. The list at the present node has this hole, and its parent will have a larger hole (see Figure 2, where the parent node of has). As in the case above, we first identify the matching push transition, and check if it agrees with the push in the last atomic hole segment in the parent. If so, we populate the witness stack i with the rightmost atomic hole segment of the parent node (see Figure 2, is populated in the stack). Each time we find a pop on backtracking the exploration tree, we find the rightmost atomic hole segment of the parent node, and keep pushing it on the stack, until we reach the node which is obtained as a result of a hole creation. Now we have completely recovered the entire hole information by backtracking, and fill the witness stack with the reversed atomic hole segments which constituted this hole. Notice that when we finish processing a hole of kind i, then the witness stack i has the hole reversed inside it, followed by a barrier. The next hole of the same kind i will be treated in the same manner.

• If the current node of the BFS tree is obtained by creating a hole of kind i in the fix-point algorithm, then we pop the contents of witness stack i till we reach a barrier. This spits out the atomic hole segments of the hole from the right to the left, giving us a sequence of push transitions, and the respective ws in between. The transitions constituting the ws are retrieved using Algorithm 5 and added. Notice that popping the witness stack i till a barrier spits out the sequence of transitions in the correct reverse order while backtracking.

5 Adding Time to Multi-pushdown systems

In this section, we briefly describe how the algorithms described in section 3 can be extended to work in the timed setting. Due to lack of space, we focus on some of the significant challenges and advances, leaving the formal details and algorithms to the supplement [11]. A TMPDA extends a MPDA with clock variables. Transitions check constraints which are conjunctions/disjunctions of constraints (called closed guards in the literature) of the form $x \leq c$ or $x \geq c$ for $c \in \mathbb{N}$ and x any clock. Symbols pushed on stacks "age" with time elapse. A pop is successful only when the age of the symbol lies within a certain interval. The acceptance condition is as in the case of MPDA.

The first main challenge in adapting the algorithms in section 3 to the timed setting was to take care of all possible time elapses along with the operations defined in Algorithm 1. The usage of closed guards in TMPDA means that it suffices to explore all runs with integral time elapses (for a proof see e.g., Lemma 4.1 in [16]). Thus configurations are pairs of states with valuations that are vectors of non-negative integers, each of which is bounded by the maximal constant in the system. Now, to check reachability we need to extend all the precomputations (transitive closure, well-nested reachability, as well as atomic and non-atomic hole segments) with the time elapse information. To do this, we use a weighted version of the Floyd-Warshall algorithm by storing time elapses during precomputations. This allows us to use this precomputed *timed* well-nested reachability information while performing the BFS tree exploration, thus ensuring that any explored state is indeed reachable by a timed run. In doing so, the most challenging part is extending the BFS tree wrt a pop. Here, we not only have to find a split of a hole into an atomic hole-segment and a hole-segment as in Algorithm 1, but also need to keep track of possible partitions of time.

Timed Witness: As in the untimed case, we generate a witness certifying non-emptiness of TMPDA. But, producing a witness for the fix-point computation as discussed earlier requires unrolling. The fix-point computation generates a pre-computed set `WRT` of tuples $((s, \nu), t, (s', \nu'))$, where $s, s' \in \mathcal{S}$, t is time elapsed in the well-nested sequence and $\nu, \nu' \in \mathbb{N}^{|\mathcal{X}|}$ are integral valuations. This set of tuples does not have information about the intermediate transitions and time-elapses. To handle this, using the pre-computed information, we define a lexicographic progress measure which ensures termination of this search.

While the details are in [11] (Algorithm 14), the main idea is as follows: the first progress measure is to check if there a time-elapse t transition possible between (s, ν) and (s', ν') and if so, we print this out. If not, $\nu' \neq \nu + t$, and some set of clocks have been reset in the transition(s) from (s, ν) to (s', ν'). The second progress measure looks at the sequence of transitions from (s, ν) to (s', ν'), consisting of reset transitions (at most the number of clocks) that result in ν' from ν. If neither the first nor the second progress measure apply, then $\nu = \nu'$, and we are left to explore the last progress measure, by exploring at most $|\mathcal{S}|$ number of transitions from (s, ν) to (s', ν'). The lexicographic progress measure seamlessly extends the witness generation to the timed setting.

6 Implementation and Experiments

We implemented a tool BHIM (**B**ounded **H**oles **I**n **M**PDA) written in C++ based on Algorithm 1, which takes an MPDA and a constant K as input and returns (*True*) iff there exists a K-hole bounded run from the start state to an accepting state of the MPDA. In case there is such an accepting run, BHIM generates one such, with minimal number of holes. For a given hole bound K, BHIM first tries to produce a witness with 0 holes, and iteratively tries to obtain a witness by increasing the bound on holes till K. In most of the cases, BHIM found the witness before reaching the bound K. Whenever BHIM's witness had K holes, it is guaranteed that there are no witnesses with a smaller number of holes.

To evaluate the performance of BHIM, we looked at some available benchmarks and modeled them as MPDA. We also added timing constraints to some examples such that they can be modeled as TMPDA. Our tests were run on a GNU/Linux system with Intel® Core™ i7–4770K CPU @ 3.50GHz, and 16GB of RAM. We considered overall 7 benchmarks, of which we sketch 3 in detail here. The details of these as well as the remaining ones are in [11].

• **Bluetooth Driver [18]**. The Bluetooth device driver example [18], has two threads and a shared memory. We model this driver using a 2-stack pushdown system, where a state represents the current valuation of the global variables and stacks are used to maintain the call-return between different functions and to keep the count of processes currently using the driver. There is also a scheduler which can preempt any thread executing a non-atomic instruction. A known error as pointed out in [18] is a race condition between two threads where one thread tries to write to a global variable and the other thread tries to read from it. BHIM found this error, with a well-nested witness. A timed extension of this example was also considered, where, a witness was obtained again with hole bound 0.

• **Bluetooth Driver v2** [10,8]. A modified version of Bluetooth driver is considered [10,8], where a counter is maintained to count the number of threads actively using the driver. A two stack MPDA models this, with one stack simulating the counter and another one scheduling the threads. Two known errors reported are (i) counter underflow where a counter goes negative, leading to some unwanted behavior of the driver, (2) interrupted I/O, where the stopping thread kills the driver while the other thread is busy with I/O. The tools SPADE and MAGIC [10,8] found one of these two errors, while BHIM found both errors, the first using a well nested witness, and the second with a 2-hole bounded witness.

• **A Multi-threaded Producer Consumer Problem**. The Producer consumer problem (see e.g., [25]) is a classic example of concurrency and synchronization. An interesting scenario is when there are multiple producers and consumers. Assume that two ingredients called 'A' and 'B' are produced in a production line in batches, where a batch can produce arbitrarily many items, but it is fixed for a day. Further, assume that (1) two units of 'A' and one unit of 'B' make an item called 'C'; (2) the production line starts by producing a batch of A's and then in the rest of the day, it keeps producing B's in batches, one after the other. During the day, 'C's are churned out using 'A' and 'B' in the proportion mentioned above and, if we run out of 'A's, we obtain an error; there is no problem if 'B' is exhausted, since a fresh batch producing 'B' is commenced. This idea can be imagined as a real life scenario where item 'A' represents an item which is very expensive to produce but can be produced in large amount but the item 'B' can be produced frequently, but it has to be consumed very soon, if it is not consumed then it becomes useless. For $m, n, k \in \mathbb{N}$, consider words of the form $a^m(b^k(c^2d)^k)^n$ where, a represents the production of one unit of 'A', b represents the production of one unit of 'B', c represents consumption of one unit of 'A' and d represents consumption of one unit of 'B'. 'm' represents the production capacity of 'A' for the day and 'k' represents production capacity of 'B'(per batch) for the day, 'n' represents the number batches of 'B' produced in a day. Unless $m \geq 2nk$, we will obtain an error. This is easily modeled using a 2 stack visibly multi pushdown automaton where a, b are push symbols of stack 1, 2 respectively and c, d are pop symbols of stack 1, 2 respectively. Let $L_{m,k,n}$ be the set of words of the above form s.t. $2nk < m$. It can be seen that $L_{m,k,n}$ does not have any well-nested word in it. The number of context switches(also, scope bound) in words of $L_{m,k,n}$ depends on the parameters k and n. However, $L_{m,k,n}$ is 2 hole-bounded : at any position of the word, the open holes come from the unmatched sequences of a and b seen so far. BHIM checked for the non-emptiness of $L_{m,k,n}$ with a witness of hole bound 2.

• **Critical time constraints [26]**. This is one of the timed examples, where we consider the language $L^{crit} = \{a^y b^z c^y d^z \mid y, z \geq 1\}$ with time constraints between occurrences of symbols. The first c must appear after 1 time-unit of the last a, the first d must appear within 3 time-units of the last b, and the last b must appear within 2 time units from the start, and the last d must appear at 4 time units. L^{crit} is accepted by a TMPDA with two timed stacks. L^{crit} has no well-nested word, is 4-context bounded, but only 2 hole-bounded.

• **A Linux Kernel bug dm_target.c [27]**. This example is about a double free bug in the file **drivers/md/dm-target.c** in Linux Kernel 2.5.71, which was introduced to fix a memory leak, but it ended up double freeing the object. BHIM found this bug with a witness of hole bound 3.

• **Concurrent Insertions in Binary Search Trees**. Concurrent insertions in binary search trees is a very important problem in database management systems. [28] proposes an algorithm to solve this problem for concurrent implementations. However, if the locks are not implemented properly, then it is possible for a thread to overwrite others. We modified the algorithm [28] to capture this bug, and modeled it as MPDA. BHIM found the bug with a witness of hole-bound 2.

• **Maze Example**. Finally we consider a robot navigating a maze, picking items; an extended (from single to multiple stack) version of the example from [17]. In the untimed setting, a witness for non-emptiness was obtained with hole-bound 0, while in the extension with time, the witness had a hole-bound 2, since the satisfaction of time constraints required a longer witness.

Results and Discussion. The performance of BHIM is presented in Table 1 for untimed examples and in Table 2 for timed examples. Apart from the results in the tables, to check the robustness of BHIM wrt parameters like the number of locations, transitions, stacks, holes and clocks (for TMPDA), we looked at

Name	Locations	Transitions	Stacks	Holes	Time Empty (mili sec)	Time Witness (mili sec)	Memory(KB)
Bluetooth	57	96	2	0	157.9	7.1	7424
Bluetooth v2(err1)	58	99	2	0	27.4	7.1	5096
Bluetooth v2(err2)	58	99	2	2	97.4	24.1	6478
MultiProdCons	11	18	2	2	11.1	0.1	1796
dm-target	13	27	2	3	42.0	5.8	4476
Binary Search Tree	29	78	2	2	60.8	5.1	5143
untimed-L^{crit}	6	10	2	2	14.9	0.7	4692
untimed-Maze	9	12	2	0	12.0	0.2	3858
L^{bh} (from Sec. 2.1)	7	13	2	2	22.2	0.6	4404

Table 1. Experimental results: Time Empty and Time Witness column represents no. of milliseconds needed for emptiness checking and to generate witness respectively.

Name	Locations	Transitions	Stacks	Clocks	c_{max}	Aged(Y/N)	Holes	Time Empty(mili sec)	Time Witness (mili sec)	Memory(KB)
Bluetooth	57	96	2	0	2	Y	0	169.9	101.3	5248
L^{crit}	6	10	2	2	8	Y	2	9965.2	3.7	203396
Maze	9	12	2	2	5	Y	2	956.8	9.7	14554

Table 2. Experimental results of timed examples. The column c_{max} is defined as the maximum constant in the automaton, and Aged denotes if the stack is timed or not

examples with an empty language, by making accepting states non-accepting in the examples considered so far. This forces BHIM to explore all possible paths in the BFS tree, generating the lists at all nodes. The scalability of BHIM wrt all these parameters are in [11].

BHIM **Vs. State of the art**. What makes BHIM stand apart wrt the existing state of the art tools is that (i) none of the existing tools handle under approximations captured by bounded holes, (ii) none of the existing tools work with multiple stacks in the timed setting (even closed guards!). The state of the art research in underapproximations wrt untimed multistack pushdown systems has produced some amazing tools like GetaFix which handles multi-threaded programs with bounded context switches. While we have adapted some of the examples from GetaFix, the latest available version of GetaFix has some issues in handling those examples[3]. Likewise, SPADE, MAGIC and the counter implementation [27] are currently not maintained. This has come in the way of a performance comparison between BHIM and these tools. Indeed, most examples handled by BHIM correspond to non-context bounded, or non scope bounded, or timed languages which are beyond Getafix. For instance, the 2-hole bounded witness found by BHIM for the language $L_{20,10}(m = 20, p = 10)$ for the multi producer consumer case cannot be found by GetaFix/MAGIC/SPADE with less than 41 context switches. In the timed setting, the Maze example (TMPDA with 2 clocks, 2 timed stacks) has a 2 hole-bounded witness where the robot visits certain locations an equal number of times. The tool [17] cannot handle this example since it handles only one stack. Lastly, [17] cannot solve binary reachability with an empty stack unlike BHIM.

BHIM **v2.** The next version of BHIM will go symbolic, inspired from GetaFix. The current avatar of BHIM showcases the efficiency of fix-point techniques extended

[3] we did get in touch with the authors, who confirmed this

to larger bounded underapproximations; indeed going symbolic will make BHIM much more robust and scalable.

Acknowledgements. We would like to thank Gennaro Parlato for the discussions we had on Getafix and for providing us benchmarks.

References

1. Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. A robust class of context-sensitive languages. In *Logic in Computer Science, 2007. LICS 2007. 22nd Annual IEEE Symposium on*, pages 161–170. IEEE, 2007.
2. Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. The language theory of bounded context-switching. In *Latin American Symposium on Theoretical Informatics*, pages 96–107. Springer, 2010.
3. Salvatore La Torre, Margherita Napoli, and Gennaro Parlato. Scope-bounded pushdown languages. *International Journal of Foundations of Computer Science*, 27(02):215–233, 2016.
4. Aiswarya Cyriac, Paul Gastin, and K Narayan Kumar. MSO decidability of multi-pushdown systems via split-width. In *International Conference on Concurrency Theory*, pages 547–561. Springer, Berlin, Heidelberg, 2012.
5. Salvatore La Torre and Margherita Napoli. Reachability of multistack pushdown systems with scope-bounded matching relations. In *International Conference on Concurrency Theory*, page 203–218. Springer, 2011.
6. Salvatore La Torre and Parlato Gennaro. Scope-bounded multistack pushdown systems: Fixed-point, sequentialization, and tree-width. 2012.
7. Salvatore La Torre, Madhusudan Parthasarathy, and Gennaro Parlato. Analyzing recursive programs using a fixed-point calculus. *ACM Sigplan Notices*, 44(6):211–222, 2009.
8. Gaël Patin, Mihaela Sighireanu, and Tayssir Touili. Spade: Verification of multithreaded dynamic and recursive programs. In *International Conference on Computer Aided Verification*, pages 254–257. Springer, 2007.
9. Shaz Qadeer. The case for context-bounded verification of concurrent programs. In *Model Checking Software, 15th International SPIN Workshop, Los Angeles, CA, USA, August 10-12, 2008, Proceedings*, pages 3–6, 2008.
10. Sagar Chaki, Edmund Clarke, Nicholas Kidd, Thomas Reps, and Tayssir Touili. Verifying concurrent message-passing C programs with recursive calls. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, page 334–349. Springer, 2006.
11. Akshay S, Gastin Paul, S Krishna, and Roychowdhury Sparsa. Supplementary material: Revisiting under-approximate reachability in MPDA. Available at `https://cse.iitb.ac.in/~sparsa/bhim/`, 2019.
12. Mohamed Faouzi Atig. Model-checking of ordered multi-pushdown automata. *arXiv preprint arXiv:1209.1916*, 2012.
13. Ahmed Bouajjani, Rachid Echahed, and Riadh Robbana. On the automatic verification of systems with continuous variables and unbounded discrete data structures. In *International Hybrid Systems Workshop*, pages 64–85. Springer, 1994.
14. Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Jari Stenman. Dense-timed pushdown automata. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, page 35–44, 2012.

15. Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Jari Stenman. The minimal cost reachability problem in priced timed pushdown systems. In *Language and Automata Theory and Applications - 6th International Conference, LATA 2012, A Coruña, Spain, March 5-9, 2012. Proceedings*, pages 58–69, 2012.
16. S. Akshay, Paul Gastin, and Shankara Narayanan Krishna. Analyzing Timed Systems Using Tree Automata. *Logical Methods in Computer Science*, Volume 14, Issue 2, May 2018.
17. S. Akshay, Paul Gastin, Shankara Narayanan Krishna, and Ilias Sarkar. Towards an efficient tree automata based technique for timed systems. In *28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany*, pages 39:1–39:15, 2017.
18. Shaz Qadeer and Dinghao Wu. Kiss: keep it simple and sequential. *Acm sigplan notices*, 39(6):14–24, 2004.
19. P Madhusudan and Gennaro Parlato. The tree width of auxiliary storage. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.
20. S. Akshay, Paul Gastin, Vincent Jugé, and Shankara Narayanan Krishna. Timed systems through the lens of logic. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, pages 1–13, 2019.
21. Aiswarya Cyriac. *Verification of communicating recursive programs via split-width. (Vérification de programmes récursifs et communicants via split-width)*. PhD thesis, École normale supérieure de Cachan, France, 2014.
22. Rajeev Alur and Parthasarathy Madhusudan. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 202–211. ACM, 2004.
23. Zhe Dang, Oscar H Ibarra, Tevfik Bultan, Richard A Kemmerer, and Jianwen Su. Binary reachability analysis of discrete pushdown timed automata. In *International Conference on Computer Aided Verification*, page 69–84. Springer, 2000.
24. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.
25. Abraham Silberschatz, Greg Gagne, and Peter B Galvin. *Operating system concepts*. Wiley, 2018.
26. Devendra Bhave, Vrunda Dave, Shankara Narayanan Krishna, Ramchandra Phawade, and Ashutosh Trivedi. A perfect class of context-sensitive timed languages. In *International Conference on Developments in Language Theory*, pages 38–50. Springer, Berlin, Heidelberg, 2016.
27. Matthew Hague and Anthony Widjaja Lin. Synchronisation- and reversal-bounded analysis of multithreaded programs with counters. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, page 260–276, 2012.
28. HT Kung and Philip L Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems (TODS)*, 5(3):354–382, 1980.

Appendix

A Details for Section 2

A.1 Proposition 1

We use the notion of Tree Terms (TTs) [17] to compute the tree-width of a given graph. Where a minimal finite set of colors are used to color the vertices and then partition the graph in two partitions such that the cut vertices are colored. The aim of this approach is to decompose a graph to "atomic" tree terms. We cannot use a color more than once in a partition of graph, unless we *forget* it. This can be modeled as a game between two player, Adam and Eve. Where, Eve's goal is to reach atomic terms with minimum finite number of colors, and Adam's goal is to make Eve's life difficult by choosing a more demanding partition.

To prove that the a model has bounded tree-width we will try to capture the runs of the model in terms of graphs (Multiply nested words [6]) and play the game mentioned above.

Tree Width of Hole Bounded Multistack Pushdown Automaton

We will capture the behaviour (any run ρ) of $K-$hole bounded multistack pushdown systems as a graph G where, every node represents a transition $t \in \Delta$ and the edge between the nodes can be of the following types.

- Linear order $\preccurlyeq$ between the transitions gives the order in which the transitions are fired in the system. We will use $\preccurlyeq^+$ to represent transitive closure of $\preccurlyeq$.
- The other type of edges represent the push pop relation between two transitions. Which means, if a transition t_1 have a push operation in the stack i and transition t_2 has the corresponding pop of the stack i, matching the push on stack i of transition t_1, then we have an edge $t_1 \curvearrowright^s t_2$ between them, which will represent the push-pop relation.

To prove that the tree width of the class of graph G is bounded, we will use coloring game [17] and show that we need bounded number of colors to split any graph $g \in G$ to atomic tree terms.

Eve will start from the right most node of the graph by coloring it. The last node of the graph can be any one of the following,

- End point of a well-nested sequence
- Pop transition t_{pp} of stack i , such that, the push t_{ps} is coming from nearest *hole* of stack i.

1. If the endpoint colored is the end point of a well-nested sequence then Eve can remove the well-nested sequence by adding another color to the first point of the well-nested sequence.
   If we look at the well-nested part, using just one more color we can split it to atomic tree terms [17].
   But the other part still remains a graph of class G so Adam will choose this partition for Eve to continue the coloring game.

2. If the last point of the graph G is a pop point t_{pp} as discussed earlier, then the corresponding push t_{ps} can come from a open *hole* or a closed *hole*.
   - If it is coming from a closed *hole* then, Eve will add color to the corresponding push t_{ps} along with the transition t_q such that, $t_{ps} \preccurlyeq^+ t_q$ and $t_{ps} - t_q$ is a well-nested sequence, which forms a atomic hole segment $(\uparrow ws)$ where, $\uparrow$ represents the push pop edge $t_{ps} \curvearrowright^s t_{pp}$ and ws represents the well-nested sequence $t_{ps} - t_q$. This operation requires 2 colors. Please note that, the right end of the *hole* which got colored after removal of $t_{ps} - t_q$ is another push of the *hole*, because *hole* are defined as a sequence $(\uparrow ws)^+$.
   - If the push is coming from open *hole* then the push transition t_{ps} is already colored from previous operation as discussed above, hence Eve will add another color $t_{q'}$ to mark the next well-nested sequence $t_{ps} - t_{q'}(ws')$ in the right of t_{ps}. Now, Eve can remove the stack edge $t_{pp} \curvearrowright t_{ps}$ along with the well-nested sequence ws'. This operation widens the *hole*.

   In both the above operations, the graph has two components one with a stack edge $t_{pp} \curvearrowright t_{ps}$ and another one with a well-nested sequence. Which require at most 1 color extra to split into atomic tree terms. On the remaining part Eve will continue playing the game from right most point.

Here, we claim that at any point of time of the coloring game, there will be $2K + 2$ active colors for $K \geq 1$ and $K \in \mathbb{N}$. Every step of the game splits the graph in two part, and one part always can be split into atomic tree terms with at most 3 colors. The remaining part will require at most 2 colors for every open *hole* in the left of the right most point of the graph. As the number of open *hole* is bounded by K, so we can not have more than K open *holes* in the left of any point. So, $2K$ colors to mark the *holes*. So, total number of colors needed to break any such graph to atomic tree terms is $2K + 4$.

A.2 Proposition 2

We describe the missing details in proposition 2.

1. L^{bh} cannot be accepted by any K-bounded phase MPDA.
   Recall that, $L^{bh} = \{a^n b^n (a^{q_i} c^{q_i+1} b^{q'_j} d^{q'_j+1})^n | n, q_i, q'_j \in \mathbb{N}\ \forall i, j \in [n]\}$, and a, b represents push in stack 1,2 respectively and c, d represents the corresponding pops from stack 1,2. For all m, consider the word $w_1 = a^m b^m (a^l c^{l+1} b^{l'} d^{l'+1})^m$. Here, clearly the number of phases is $K = 2m$. Now if w_1 is accepted by some phase bounded MPDA M then it must have $2m$ as the bound on the phases which will not be sufficient to accept $w_2(a^{m+1} b^{m+1} (a^l c^{l+1} b^{l'} d^{l'+1})^{m+1}) \in L^{bh}$.
2. $L' = \{(ab)^n c^n d^n \mid n \in \mathbb{N}\}$ cannot be accepted by any K-hole bounded MPDA.
   For any $m \in \mathbb{N}$ assume a word $w_1 = (ab)^m c^m d^m \in L'$, where a, b represents push in stack 1,2 respectively and c, d represents the corresponding pops from stack 1,2. Clearly, this can be accepted by a bounded *hole* multistack pushdown automata M with bound $= 2m$. Now if L' is accepted by M then

it must also accept, $w_2 = (ab)^{m+1}c^{m+1}d^{m+1}$. However, the number of *holes* required to accept w_2 is $2(m+1) > 2m$. This contradicts the assumption that M accepts the language.

B Details for Section 3

In this section, we provide all the subroutines mentioned in Section 3 and used in Algorithm 1 for MPDA.

We start by presenting Algorithm 2 which computes the well-nested reachability relation, i.e., it computes the set WR of all pairs of states (s, s') such that there is a well-nested sequence from s to s'. The proof of correctness of this algorithm

Algorithm 2: Well Nested Reachability

1 **Function** `WellNestedReach`($M = (\mathcal{S}, \Delta, s_0, \mathcal{S}_f, n, \Sigma, \Gamma)$)**:**
  Result: WR$:= \{(s, s')|s'$ is reachable from s via a well-nested sequence $\}$
2   $\mathcal{R}_c := \{(s, s)|s \in \mathcal{S}\}$;
3   **forall** $(s_1, \mathsf{op}, a, s_2) \in \Delta$ *with* $\mathsf{op} = \mathsf{nop}$ **do**
4     $\mathcal{R}_c := \mathcal{R}_c \cup \{(s_1, s_2)\}$; \\`Transitions with nop operation`
5   $\mathcal{R}_c :=$ `TransitiveClosure`$(\mathcal{R}_c)$; \\`Using Floyd-Warshall Algorithm`
6   **while** *True* **do**
7   WR$:= \mathcal{R}_c$;
8   **forall** $(s, \downarrow_i(\alpha), a, s_1) \in \Delta$ **do**
9   **forall** $(s_1, s_2) \in$ *WR* **do**
10   **forall** $(s_2, \uparrow_i(\alpha), a, s') \in \Delta$ **do**
11          $\mathcal{R}_c := \mathcal{R}_c \cup \{(s, s')\}$; \\`Wrap well-nested sequence with matching push-pop`
12     $\mathcal{R}_c :=$ `TransitiveClosure`$(\mathcal{R}_c)$;
13   **if** $\mathcal{R}_c \setminus$ *WR* $= \emptyset$ **then**
14   break; \\`Break when no new well-nested sequence added`
15 **return** *WR***;**

(and thus Lemma 1) is easy to see. First, line 5 the set $\mathcal{R}_c$ contains all pairs $(s,')$ such that s' is reachable from s in the MPDA without using the stack. Then for every push transition from a state s we check in lines 8-11 whether there is an (already computed) well-nested sequence that can reach a state s' with a corresponding pop transition and if so we add (s, s'). We take the transitive closure and repeat this process, hence guaranteeing that at fixed point we will have all well-nested pairs, i.e., WR.

Details of Algorithm 3 For a given list μ Algorithm 3 tries to extend the list μ by adding a hole of a stack i. This is achieved by checking the last state s_{last} the list μ and finding all possible hole in HS_i that start with s_{last} and appending the hole followed by a suitable well-nested sequence to μ.

Algorithm 3: AddHole

Function `AddHole`$_i(\mu, HS_i)$**:**
  Result: Set, a set of lists.
  $Set := \emptyset$;
  forall $(i, s, s') \in HS_i$ *with* $s = \mathsf{last}(\mu)$ **do**
    $\mu' := copy(\mu)$; \\`Create a copy of the list` μ
    $\mathsf{trunc}(\mu')$; \\`trunc(`μ`) is defined as remove(last(`μ`)))`
    $\mu'.\mathsf{append}[(i, s, s'), s']$; \\`Append to the list` μ'
    μ'.`NumberOfHoles` $:= \mu$.`NumberOfHoles` $+ 1$;
    $Set := Set \cup \{\mu'\}$;
return Set;

Algorithm 4: Extend with a pop

Function `AddPop`$_i(\mu, M = (\mathcal{S}, \Delta, s_0, \mathcal{S}_f, n, \Sigma, \Gamma), AHS_i, HS_i, \mathtt{WR})$**:**
  Result: Set, a set of lists
  $Set := \emptyset$;
  $(i, s_1, s_3) := lastHole_i(\mu)$; \\`Get the last open hole of stack` i
  forall $(i, s_1, s_2) \in HS_i$, $(s_2, \alpha, s_3) \in AHS_i$, $(s, \uparrow_i(\alpha), s') \in \Delta$, $s = \mathsf{last}(\mu)$ *and* $(s', s'') \in \mathtt{WR}$ **do**
    $\mu' := copy(\mu)$;
    $\mathsf{trunc}(\mu')$;
    $\mu'.\mathsf{append}(s'')$;
    if $(s_1 = s_2)$ **then**
      $\mu'' := copy(\mu)$;
      $\mathsf{trunc}(\mu'')$;
      $\mu''.\mathsf{append}(s'')$;
      $\mu''.\mathsf{remove}((i, s_1, s_3))$; \\`Remove the hole` (i, s_1, s_2) `from the list` μ''
      μ''.`NumberOfHoles` $:= \mu$.`NumberOfHoles`-1;
      $Set := Set \cup \{\mu''\}$;
    $\mu'.\mathsf{replace}((i, s_1, s_3)$, by $(i, s_1, s_2))$; \\`Replace bigger hole` (i, s_1, s_3) `by new smaller hole` (i, s_1, s_2)
    $Set := Set \cup \{\mu'\}$;
return Set;

Details of Algorithm 4 For a given list μ this algorithm tries to extend μ with a pop operation. The algorithm starts with extracting the last hole(H_i) of stack i. Due to the well-nested property, the pop (which is not part of a well-nested sequence) must be matched with the first pending push in the last hole of stack i in μ. Then the algorithm checks for all atomic hole-segments AHS_i and hole-segments HS_i s of the stack i, such that, the hole H_i can be partitioned in HS_i and AHS_i. Then the push in AHS_i is matched with the matched pop operation and the hole is now shrunk into HS_i. So, the algorithm replaces H_i with HS_i. If the H_i is same as some AHS_i then, the hole can be closed and hence it removes the hole from the list. In this case it also reduces the count of the number of holes in the list. Note that without the pre-computation of AHS_i and HS_i this part of the algorithm is fairly difficult. Using the pre-computation allow us to use simple table look ups when the states are known, this takes only constant time.

C Details for Section 4

The algorithm for witness generation, as discussed in the main part of the paper, does a backtracking on the BFS tree. When we encounter a node in the BFS tree extending the list with a pop, creating a hole, we use the last state in the list, the transition information from the node, and the witness stack for backtracking. During the backtracking we also need to know the sequence of transitions responsible for the well-nested sequences, which can be generated using the Algorithm 5. The backtracking Algorithm 6 is discussed in the following example.

Algorithm 5: Well-nested witness generation for MPDA

Function `Witness`$(s_1, s_2, M = (\mathcal{S}, \Delta, s_0, \mathcal{S}_f, n, \Sigma, \Gamma), \texttt{WR})$:
    Result: A sequence of transitions for a run resulting the well-nested sequence `WR`
    if $s_1 == s_2$ **then**
        return ϵ;
    if $\exists t = (s_1, \mathsf{nop}, a, s_2) \in \Delta$ **then**
        return t;
    forall $s', s'' \in \mathcal{S}$ **do**
        if $((s_1 \neq s') \vee (s'' \neq s_2)) \wedge (s', s'') \in \texttt{WR} \wedge \exists t = (s_1, \downarrow_i (\alpha), a, s') \in \Delta \wedge \exists t_2 = (s'', \uparrow_i (\alpha), a', s_2) \in \Delta$ **then**
            path=`Witness`$(s', s'', M, \texttt{WR})$;
            return $t.path.t_2$;
    forall $s \in \mathcal{S}$ **do**
        if $(s \neq s_1 \vee s \neq s_2) \wedge (s, s_1) \in \texttt{WR} \wedge (s, s_2) \in \texttt{WR}$ **then**
            path1=`Witness`$(s_1, s, M, \texttt{WR})$;
            path2 = `Witness`$(s, s_2, M, \texttt{WR})$;
            return *path1.path2*;

An Illustrating Example for Witness Generation

We illustrate the multistack case on an example. Note that in figures illustrating examples, we use colored uparrows and downarrows with subscript for stacks, and a superscipt i representing the ith push or pop of the relevant colored stack.

Assume that the path we obtain on back tracking is the reverse of Figure 3. Holes arising from pending pushes of stack 1 are red holes, and those from stack 2 are blue holes in the figure. We have two red holes: the first red hole has a left end point $\downarrow_1^1$, and right end point ws_3. The second red hole has a left end point

Algorithm 6: Non-well-nested witness generation for MPDA

Function `HoleWitness`($\mu, M = (\mathcal{S}, \Delta, s_0, \mathcal{S}_f,\ n, \Sigma, \Gamma)$,*WR*,$AHS_i$,$HS_i$):

  Result: A sequence of transitions for an accepting run

  global WitnessStacks $= \{St_i \mid i \in [n]\}$; \\`Witness stacks for every stack i`

  $\mu_p = Parent(\mu)$; \\`Parent function returns the parent node of` μ `in the BFS exploration tree`

  $op_\mu = ParentOp(\mu)$; \\`ParentOp function returns the operation that extends` $Parent(\mu)$ `to` μ `in the BFS exploration tree`

  if $op_\mu == ExtendByPop_i(\uparrow_i \alpha.wr_{pop}) \wedge wr_{pop} \in$ *WR* **then**

    $(i, s_1, s_2) = lastHole_i(\mu_p)$;

    if $(s_i, \alpha, s_2) \in AHS_i \wedge (s_1, \alpha, s_2) =\downarrow_i (\alpha).wr_{push} \wedge wr_{push} \in$ *WR* **then**

      $push(St_i, \#)$;

      $list =$ `Witness`(wr_{push});

      $\forall t \in list, push(St_i, t)$;

      $push(St_i, \downarrow_i (\alpha))$;

      $list_{pop} =$ `Witness`(wr_{pop});

      return `HoleWitness`$(\mu_p).\uparrow_i (\alpha).list_{pop}$;

    else if $(s_i, \alpha, s_2) \notin AHS_i \wedge (i, s_i, s_2) = (s_i, \alpha, s_3).(i, s_3, s_2) \wedge (s_1, \alpha, s_3) \in AHS_i \wedge (i, s_3, s_2) \in HS_i \wedge (s_1, \alpha, s_3) =\downarrow_i (\alpha).wr_{push} \wedge wr_{push} \in$ *WR* **then**

      $list =$ `Witness`(wr_{push});

      $\forall t \in list, push(St_i, t)$;

      $push(St_i, \downarrow_i (\alpha))$;

      $list_{pop} =$ `Witness`(wr_{pop});

      return `HoleWitness`$(\mu_p).\uparrow_i (\alpha).list_{pop}$;

  if $op_\mu == ExtendByHole_i$ **then**

    $list = \epsilon$;

    while $pop(St_i) \neq \#$ **do**

      $list = list.pop(St_i)$;

    return `HoleWitness`$(\mu_p).list$;

$\downarrow_1^4$, and right end point $\downarrow_1^5$. The blue hole has left end point $\downarrow_2^1$ and right end point ws_4.

1. From the final configuration s_f, on backtracking, we obtain the pop operation ($\uparrow_1^1$). By the fixed-point algorithm, this operation closes the first red hole, matching the first pending push $\downarrow_1^1$. In the BFS exploration tree, the parent node has the red atomic hole consisting of just the $\downarrow_1^1$. Notice also that, in the parent node, this is the only red hole, since the second red hole in Figure 3 is closed, and hence does not exist in the parent node. We use two witness stacks, a red witness stack and a blue witness stack to track the information with respect to the red and blue holes. On encountering a pop transition closing a red hole, we populate the red witness stack with (i) a barrier signifying closure of a red hole, and (ii) the matching push transition $\downarrow_1^1$.

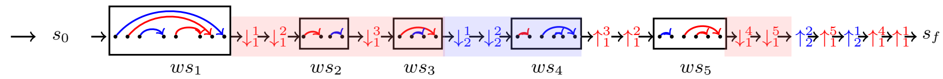


Fig. 3. A run with 3 holes. The blue hole corresponds to the blue stack and the red holes to the red stack. A final state is reached from $\uparrow_1^1$ on a discrete transition.

2. Continuing with the backtracking, we obtain the pop operation $\uparrow_1^4$, which, by the fixed-point algorithm, closes the second red hole. In the parent node, we have the atomic red hole consisting of just the $\downarrow_1^4$. The red witness stack contains from bottom to top, $\#\downarrow_1^1$. Since we encounter a closure of a red hole again, we push to the red witness stack, $\#\downarrow_1^4$. This gives the content of the red witness stack as $\#\downarrow_1^1\#\downarrow_1^4$ from bottom to top. The next pop transition $\uparrow_2^1$ is processed the same way, populating the blue witness stack with $\#\downarrow_2^1$.
3. Continuing with backtracking, we have the pop transition $\uparrow_1^5$. Since this is not closing the second red hole, but only shrinking it, we push $\downarrow_1^5$ on top of the red witness stack (no barrier inserted). This gives the content of the red witness stack as $\#\downarrow_1^1\#\downarrow_1^4\downarrow_1^5$.
4. We next have the pop transition $\uparrow_2^2$, which by the fixed-point algorithm, shrinks the blue hole. The parent node has the blue hole with left end point $\downarrow_2^1$, and ends with the atomic hole segment $\downarrow_2^2 ws_4$. We push onto the blue witness stack, this atomic hole obtaining the witness stack contents (bottom to top) $\#\downarrow_2^1\downarrow_2^2 ws_4$.
5. In the next step of backtracking, we are at a parent node using the create hole operation (creation of the second red hole). We pop the contents of the red witness stack till we hit a #, giving us the transitions $\downarrow_1^5\downarrow_1^4$ in the reverse order.
6. Next, on backtracking, we encounter the pop operation $\uparrow_1^2$ along with a well-nested sequence ws^5. We retrieve from this information, ws^5, and using the Algorithm 5, obtain the sequence of transitions constituting ws^5. The parent node has a hole segment with left end point $\downarrow_1^1$, followed by the atomic hole segment $\downarrow_1^2 ws_2$. We find the matching push transition as $\downarrow_1^2$, and push the last atomic hole segment to the red witness stack, obtaining witness stack contents $\#\downarrow_1^1\downarrow_1^2 ws_2$. The next pop operation $\uparrow_1^3$ leads us to the next parent having a hole with left end point $\downarrow_1^1$, and ending with the atomic hole $\downarrow_1^3 ws_3$. We push this to the red witness stack obtaining $\#\downarrow_1^1\downarrow_1^2 ws_2\downarrow_1^3 ws_3$ as the stack contents from bottom to top.
7. Next, the backtracking leads us to the parent creating the blue hole. We pop the blue witness stack retrieving ws_4 followed by the push transitions $\downarrow_2^2$ and $\downarrow_2^1$. The transitions of ws_4 are obtained from Algorithm 5.
8. Continuing with the backtracking, we arrive at the transition which creates the first red hole. At this time, we pop the red witness stack until we hit a barrier. We obtain ws_3, and then we retrieve the transition $\downarrow_1^3$, followed by ws_2, and the push transitions $\downarrow_1^2$ and $\downarrow_1^1$. Transitions of ws_3, ws_2 are retrieved using Algorithm 5.

9. Further backtracking leads us to the parent obtained by extending with the well-nested sequence ws_1. We retrieve the transitions in ws_1 using Algorithm 5. The last backtracking lands us at the root $[s_0]$ and we are done.

D Details for Section 5

This part of the appendix is devoted to extending our algorithms for reachability and witness generation. We start by defining timed multistack push down automata. Then, Appendix E details the (binary) reachability and algorithms therein, whereas Appendix F describes the generation of a witness for TMPDA.

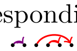
Timed Multi-stack Pushdown Automata (TMPDA)

For $N \in \mathbb{N}$, we denote the set of numbers $\{1, 2, 3 \cdots N\}$ as $[N]$. $\mathcal{I}$ denotes the set of closed intervals $\{I | I \subseteq \mathbb{R}_+\}$, such that the end points of the intervals belong to $\mathbb{N}$. $\mathcal{I}$ also contains a special interval $[0, 0]$. We start by defining the model of timed multi-pushdown automata.

Definition 3. *A Timed Multi-pushdown automaton (*TMPDA *[16]) is a tuple $M = (\mathcal{S}, \Delta, s_0, \mathcal{S}_f, \mathcal{X}, n, \Sigma, \Gamma)$ where, $\mathcal{S}$ is a finite non-empty set of locations, Δ is a finite set of transitions, $s_0 \in \mathcal{S}$ is the initial location, $\mathcal{S}_f \subseteq \mathcal{S}$ is a set of final locations, $\mathcal{X}$ is a finite set of real valued variables known as clocks, n is the number of (timed) stacks, Σ is a finite input alphabet, and Γ is a finite stack alphabet which contains $\perp$. A transition $t \in \Delta$ can be represented as a tuple $(s, \varphi, \mathsf{op}, a, R, s')$, where, $s, s' \in \mathcal{S}$ are respectively, the source and destination locations of the transition t, φ is a finite conjunction of closed guards of the form $x \in I$ represented as, $(x \in I' \wedge y \in I'' \dots)$ for $x, y \in \mathcal{X}$ and $I', I'' \in \mathcal{I}$, $R \subseteq \mathcal{X}$ is the set of clocks that are reset, $a \in \Sigma$ is the label of the transition, and* op *is one of the following stack operations (1)* nop*, or no stack operation, (2) $(\downarrow_i \alpha)$ which pushes $\alpha \in \Gamma$ onto stack $i \in [n]$, (3) $(\uparrow_i^I \alpha)$ which pops stack i if the top of stack i is $\alpha \in \Gamma$ and the time elapsed from the push is in the interval $I \in \mathcal{I}$.*

A *configuration* of TMPDA is a tuple $(s, \nu, \lambda_1, \lambda_2, \dots, \lambda_n)$ such that, $s \in \mathcal{S}$ is the current location, $\nu \colon \mathcal{X} \to \mathbb{R}$ is the current clock valuation and $\lambda_i \in (\Gamma \times \mathbb{R})^*$ represents the current content of i^{th} stack as well as the *age* of each symbol, i.e., the time elapsed since it was pushed on the stack. A pair (s, ν), where s is a location and ν is a clock valuation is called a *state*.

The semantics of the TMPDA is defined as follows: a run σ is a sequence of alternating time elapse and discrete transitions from one configuration to another. The time elapses are non-negative real numbers, and, on discrete transitions, the valuation ν of the current configuration is checked to see if the clock constraints are satisfied; likewise, on a pop transition, the age of the symbol popped is checked. Projecting out the operations of a single stack from σ results in a well-nested sequence. A run is accepting if it starts from the initial state with all clocks set to 0, and reaches a final state with all stacks empty. The language accepted by a TMPDA is defined as the set of timed words generated by the accepting runs of the TMPDA. Since the reachability problem for TMPDA is Turing complete (this is the case even without time), we consider under-approximate reachability.

A sequence of transitions is said to be **complete** if each push has a matching pop and vice versa. A sequence of transitions is said to be **well-nested**, denoted ws, if it is a sequence of nop-transitions, or a concatenation of well-nested sequences $ws_1 ws_2$, or a well-nested sequence surrounded by a matching push-pop pair $(\downarrow_i \alpha)$ ws $(\uparrow_i^I \alpha)$. If we visualize this by drawing edges between pushes and their corresponding pops, well-nested sequences have no crossing edges, as in  and , where we have two stacks, depicted with red and violet edges. We emphasize that a well-nested sequence can have well-nested edges from any stack. In a sequence σ, a push (pop) is called a **pending** push (pop) if its matching pop (push) is not in the same sequence σ. For TMPDA every sequence also carries total time elapsed during the sequence, this is helpful to check stack constraints, and it is sufficient to store time till the maximum stack constraint, i.e., the maximum constant value that appeared in the stack constraints.

Tree Width of Bounded Hole TMPDA

We will capture the behaviour(any run ρ) of $K-$hole bounded multistack pushdown systems as a graph G where, every node represents a transition $t \in \Delta$ and the edge between the nodes can be of three types.

- Linear order($\preccurlyeq$) between the transition which gives the order in which the transitions are fired. We will use $\preccurlyeq^+$ to represent transitive closure of $\preccurlyeq$.
- Timing relations $\curvearrowright^{c \in I} \in \preccurlyeq^+ \forall c \in \mathcal{X}$ and $I \in \mathcal{I}$ such that, $t_1 \curvearrowright^{c \in I} t_2$ if and only if the clock constraint $c \in I$ is checked in the transition t_2 and $t_1 \preccurlyeq^+ t_2$ has the latest reset of clock c with respect to t_2.
- The other type of edges represent the push pop relation between two transitions. Which means, if a transition t_1 have a push operation in any one of the stack i and transition t_2 has pop transition of the stack i which matches with the push transition at t_1, then we have an edge $t_1 \curvearrowright^s t_2$ between them, which will represent the stack edge.

To prove that the tree width of the class of graph G is bounded, we will use coloring game [17] and show that we need bounded number of colors to split any graph $g \in G$ to atomic tree terms.

Eve will start from the right most node of the graph by coloring it. The last node of the graph can be any one of the following,

- End point of a well-nested sequence
- Pop transition t_{pp} of stack i , such that, the push t_{ps} is coming from nearest *hole* of stack i.

1. If the end point colored is the end point of a well-nested sequence then Eve can remove the well-nested sequence by adding another color to the first point of the well-nested sequence. But, there may be some transitions t in the well-nested sequence with clock constraints $c \in I$ such that, the recent reset

of the clock c, with respect to t is in the left of the well nested sequence. In order to remove the well-nested sequence she have to color the nodes which represent the transitions with recent reset points of the clocks $c \in \mathcal{X}$. This step require at most $|\mathcal{X}|$ colors. Now, she can split the graph in two parts, one of them will be well-nested with two end points colored. Also, the clock constraint edges, which are coming from the left of the well-nested sequence are hanging in the left, are colored. There can be at most $|\mathcal{X}|$ hanging colored points possible in the left of the well-nested sequence. The other part will be the remaining graph with the right most point colored along with the colored recent reset points on the left of right most colored point. which are also the hanging points of the previous partition.
If we look at the well-nested part with hanging clock edges, using just one more color we can split it to atomic tree terms [17].
But the other part still remains a graph of class G so Adam will choose this partition for Eve to continue the coloring game.
2. If the last point of the graph G is a pop point t_{pp} as discussed earlier, then the corresponding push t_{ps} can come from a open *hole* or a closed *hole*.
   - If it is coming from a closed *hole* then, Eve will add color to the corresponding push t_{ps} along with the transition t_q such that, $t_{ps} \preccurlyeq^+ t_q$ and $t_{ps} - t_q$ is a well-nested sequence, which forms a atomic hole segment $(\uparrow ws)$ where, $\uparrow$ represents the push pop edge $t_{ps} \curvearrowright^s t_{pp}$ and ws represents the well-nested sequence $t_{ps} - t_q$. But just as we discussed in previous scenario of removing well-nested sequence, there may be some clock constraint $c \in \mathcal{X}$ in the well-nested sequence ws such that the transition with the recent resets are from the left of $(\uparrow ws)$ and without coloring them Eve can not remove the $(\uparrow ws)$. Similarly, there may be some clock resets inside $\uparrow ws$ from which there are clock constraint edges are going to the right of $\uparrow ws$. Eve has to color all those points inside the $\uparrow ws$ which corresponds to those clock reset points in $\uparrow ws$. So, she have to color at most $2|\mathcal{X}|$ reset points to remove the stack edge $t_1 \curvearrowright t_2$ along with the well-nested sequence $t_{ps} - t_q(\uparrow ws)$, which makes the closed *hole* open with colors in both ends of hole and at most $|\mathcal{X}|$ colors in the left of the hole and at most $|\mathcal{X}|$ colored hanging points inside the *hole*. This operation requires $2 + 2|\mathcal{X}|$ more colors. Please note that, the right end of the *hole* which got colored after removal of $t_{ps} - t_q$ is another push of the *hole*, because *hole* are defined as a sequence $(\uparrow ws)^+$.
   - If the push is coming from open *hole* then the push transition t_{ps} must be colored from previous operation as discussed above, hence Eve will add another color $t_{q'}$ to mark the next well-nested sequence $t_{ps} - t_{q'}(ws')$ in the right of t_{ps}. But, similar to above section here also there may be some clock resets of clock $i \in \mathcal{X}$ inside the ws' which is being checked in the right of the ws'. These reset points can be at most $|\mathcal{X}|$ and needs $|\mathcal{X}|$ colors. Now, Eve can remove the stack edge $t_{pp} \curvearrowright t_{ps}$ along with the well-nested sequence ws'. This operation widens the *hole*. Note that at any point of the game, hanging clock reset points inside the *hole* and in left side of hole is bounded by $|\mathcal{X}|$. This operation requires at most

$1 + |\mathcal{X}|$ colors but subsequent application of this operation can reuse colors.

In both the above operations, we can split the graph in two parts, one with a stack edge $t_{pp} \curvearrowright t_{ps}$ and a well-nested sequence, with at most $|\mathcal{X}|$ hanging points for each clock in the left of the t_{pp} and at most $|\mathcal{X}|$ colors inside the ws. which require at most 1 color to split into atomic tree terms without any extra colors. On the remaining part Eve will continue playing the game from right most point.

Here, we claim that at any point of time of the coloring game, there will be $2K + (2K+1)|\mathcal{X}| + 2$ active colors for $K \geq 1$ and $K \in \mathbb{N}$. Every step of the game splits the graph in two part, and one part always can be split into atomic tree terms with at most $2|\mathcal{X}| + 3$ colors. The remaining part will require $2 + 2|\mathcal{X}|$ colors for every open *hole* in the left of the right most point of the graph. As the number of open *hole* is bounded by K, so we can not have more than K open *holes* in the left of any point. So, $2K + 2K|\mathcal{X}|$ colors to mark the *holes*, $1 + |\mathcal{X}|$ for the right most point and recent reset points with respect to the right most point, $1 + |\mathcal{X}|$ for coloring the well-nested sequence after a matched push and the possible reset points inside the well-nested sequence, but we will need to color such well-nested sequence once at any point of time, which gives a total color of $2K(|\mathcal{X}| + 1) + 2(|\mathcal{X}| + 1) = (2K + 2)(|\mathcal{X}| + 1)$.

E Reachability in TMPDA

In this section, we discuss how the BFS tree exploration extends in the timed setting. To begin, we talk about how a list at any node in the tree looks like.

Representation of Lists for BFS Tree

Each node of the BFS tree stores a list of bounded length. A list is a sequence of states (s, ν) separated by time elapses (t), representing a K-hole bounded run in a concise form. The simplest kind of list is a single state (s, ν) or a well-nested sequence (s, ν, t, s_i, ν_i) with time elapse t. Note that because of time constraints we need to store total time elapsed to reach one state from another. This is why we are keeping a time stamp between two states. Recall, the hole in MPDA is defined as a tuple (i, s, s'). For TMPDA we need to store total time elapsed in the hole as well, so it can be represented as a tuple $H = (i, s, \nu, s', \nu', t_h)$, where, t_h is the time elapse in the hole and $(s, \nu), (s', \nu')s'$ being the end states of the hole. Also, the maximum possible value of time stamp is bounded by the maximum integer value in the constraints (both pop and clock). So, the total possible values that the variable t_i can take is also bounded. Let H, t represent respectively holes (of some stack) and time elapses. A list with holes has the form $(s_0, \nu_0).t.(H)^*(H.t.(s', \nu'))$. For example, a list with 3 holes of stacks i, j, k is

$$[(s_0, \nu_0), t_1, (i, s_1, \nu_1, s_2, \nu_2, t_2), t_3, (j, s_3, \nu_3, s_4, \nu_4, t_4), t_5, (k, s_5, \nu_5, s_6, \nu_6, t_6), t_7, (s_7, \nu_7)]$$

Algorithm 7: Algorithm for Emptiness Checking of hole bounded TMPDA

Function `IsEmptyTimed`($M = (\mathcal{S}, \Delta, s_0, \mathcal{S}_f, \mathcal{X}, n, \Sigma, \Gamma), K$):
  Result: True or False
  `WRT` := `WellNestedReachTimed`(M); \\Solves binary reachability for pushdown system
  if *some* $(s_0, \nu_0, t, s_1, \nu_1) \in$ *WRT with* $s_1 \in \mathcal{S}_f$ **then**
    return *False*;
  forall $i \in [n]$ **do**
    $AHST_i := \emptyset$;
    forall $(s, \phi, \downarrow_i(\alpha), \rho, a, s_1) \in \Delta,\ \nu \models \phi,$ *and* $\nu_1 = \rho[\nu]$ **do**
      forall $(s_1, \nu_1, t, s', \nu') \in$ *WRT* **do**
        $AHST_i := AHST_i \cup (i, s, \nu, \alpha, s', \nu', t)$;
    $Set_i := \{(s, \nu, t, s', \nu') \mid \exists \alpha(i, s, \nu, \alpha, s', \nu', t) \in AHS_i\}$;
    $HS_i := \{(i, s, \nu, s', \nu', t) \mid (s, \nu, t, s', \nu') \in$ `TransitiveClosure`$(Set_i)\}$;
  $\mu := [s_0, \nu_0]$;
  μ.`NumberOfHoles` := 0;
  `SetOfLists`$_{new}$:= $\{\mu\}$, `SetOfLists`$_{old}$:= $\emptyset$;
  while *SetOfLists*$_{new}$ $\setminus$ *SetOfLists*$_{old}$ $\neq \emptyset$ **do**
    `SetOfLists`$_{diff}$:= `SetOfLists`$_{new}$ $\setminus$ `SetOfLists`$_{old}$;
    `SetOfLists`$_{old}$:= `SetOfLists`$_{new}$;
    forall $\mu' \in$ *SetOfLists*$_{diff}$ **do**
      if μ'.`NumberOfHoles` $< K$ **then**
        forall $i \in [n]$ **do**
          `SetOfLists`$_h$:= `AddHoleTimed`$_i$ (μ', HST_i); \\Add hole for stack i
          forall $\mu_2 \in$ *SetOfLists*$_h$ **do**
            `SetOfLists`$_{new}$:= `SetOfLists`$_{new}$ $\cup\ \mu_2$;
      if μ'.`NumberOfHoles` > 0 **then**
        forall $i \in [n]$ **do**
          `SetOfLists`$_p$:= `AddPopTimed`$_i$ $(\mu', M, AHST_i, HST_i,$`WRT`$)$; \\Add pop for stack i
          forall $\mu_3 \in$ *SetOfLists*$_p$ **do**
            if $\mu_3.last \in \mathcal{S}_f$ *and* μ_3.`NumberOfHoles` $= 0$ **then**
              return *False*; \\If reached destination state
            `SetOfLists`$_{new}$:= `SetOfLists`$_{new}$ $\cup\ \mu_3$;
  return *True*;

Algorithm 8: States

Function `States`($M = (\mathcal{S}, \Delta, s_0, \mathcal{S}_f, \mathcal{X}, n, \Sigma, \Gamma)$):
  Result: F
  $F := \{(s, \nu) \mid \forall s \in \mathcal{S} \wedge \forall c \in \mathcal{X}, \nu[c] \leq max(c) + 1\}$;
return F;

Algorithms for TMPDA

The function TimeElapse returns the states which are reachable from the state (s_1, ν_1) via time elapse. It also stores the total time elapsed to reach the state. This function is only useful for timed systems.

Algorithm 9: Time Elapse

Function `TimeElapse(`(s_1, ν_1)`)`**:**
  Result: Set
  $Set := \emptyset$;
  $t := 0$;
  while $t \leq c_{max}$ **do**
    $\forall i \in X : \nu_2[i] := \mathtt{Min}(\nu_1[i] + t, c_i)$;
    $Set := Set \cup (s_1, \nu_1, t, s_1, \nu_2)$;
    $t := t + 1$;
return Set**;**

Algorithm 10: Well Nested Reach Timed

Function `WellNestedReachTimed(`$M = (\mathcal{S}, \Delta, s_0, \mathcal{S}_f,\ \mathcal{X}, n, \Sigma, \Gamma)$`)`**:**
  Result: `WRT` $:= \{(s, \nu, t, s', \nu') | (s', \nu')$ is reachable from (s, ν) by time elapse t via a well-nested sequence$\}$
  F = `States(`M`)`;
  Set = $\{(s, \nu, p, s, \nu) \mid (s, \nu) \in F\}$;
  forall $(s, \nu) \in F$ **do**
    $Set = Set \cup$`TimeElapse(`(s, ν)`)`;
    forall $(s, \varphi, \mathsf{nop}, a, R, s') \in \Delta$ *with* $\nu \models \phi$ **do**
      $Set := Set \cup (s, \nu, 0, s', R[\nu])$
  $\mathcal{R}_{tc}$= `TransitiveClosureTimed(`Set`)`;
  while *True* **do**
    `WRT` $:= \mathcal{R}_{tc}$;
    forall $(s, \phi_1, \downarrow_i(\alpha), \rho_1, a, s_1) \in \Delta$ *and* $(s, \nu) \in F$ *with* $\nu \models \phi_1$ **do**
      forall $(s_1, \rho_1[\nu], t, s_2, \nu_2) \in \mathcal{R}_{tc}$ **do**
        forall $(s_2, \phi_2, \uparrow_i^I(\alpha), \rho_2, a, s') \in \Delta$ *with* $\nu_2 \models \phi_2,\ t \in I$ **do**
          $\mathcal{R}_{tc} := \mathcal{R}_{tc} \cup (s, \nu, t, s', \rho_2[\nu_2])$;
    $\mathcal{R}_{tc}$:= `TransitiveClosureTimed(`$\mathcal{R}_{tc}$`)`;
    if $\mathcal{R}_{tc} \setminus \mathit{WRT} = \emptyset$ **then**
      break;
return WRT**;**

Algorithm 11: Add Hole Timed

Function `AddHoleTimed`$_i$`(`μ, HST_i`)`**:**
  Result: $Set = \{\ \mu |\ \mu$ is a list of states and time elapses$\}$
  $Set := \emptyset$;
  $(s, \nu) := \mathsf{last}(\mu)$;
  forall $(i, s, \nu, t, s', \nu') \in HST_i$ **do**
    $\mu' = copy(\mu)$;
    $\mathsf{trunc}(\mu')$;   /* $\mathsf{trunc}(\mu)$ is defined as $\mathsf{remove}(\mathsf{last}(\mu))$ */
    $\mu'.\mathsf{append}[(i, s, \nu, t, s', \nu'), 0, (s', \nu')]$;
    μ'.`NumberOfHoles` $:= \mu$.`NumberOfHoles` $+ 1$;
    $Set := Set \cup \{\mu'\}$;
return Set**;**

Algorithm 12: Extend with a pop Timed

Function `AddPopTimed`$_i(\mu, M = (\mathcal{S}, \Delta, s_0, \mathcal{S}_f, \mathcal{X}, n, \Sigma, \Gamma), AHST_i, HST_i,$ `WRT`$)$**:**

  Result: $Set = \{\ \mu \mid \mu$ is a list of states and time elapses$\}$

  $Set := \emptyset$;

  $[t_l, (s, \nu)] :=$ last(μ);

  $[t', (i, s_1, \nu_1, t, s_3, \nu_3), t''] := lastHole_i(\mu)$;

  $t_3 :=$ The sum of the time elapses in the list μ between $(s_2, \nu_2)_{R_i}$ and (s, ν);

  forall $(i, s_1, \nu_1, t_1, s_2, \nu_2) \in HST_i,\ (i, s_2, \nu_2, t_2, \alpha, s_3, \nu_3) \in AHST_i,$ $(s, \phi, R, \uparrow_i^I(\alpha), s') \in \Delta$ *with* $t = t_1 + t_2,\ \nu \models \phi$ *and* $t_2 + t_3 \in I,$ *and* $(s', R[\nu], t_4, s'', \nu'') \in$ `WRT` **do**

    $\mu' = copy(\mu)$;

    trunc(μ');

    μ'.append$([t_l \oplus t_4, (s'', \nu'')]$;

    μ'.replace$([t', (i, s_1, \nu_1, t, s_3, \nu_3), t''],$
      $[t', (i, s_1, \nu_1, t_1, s_2, \nu_2), t_2 \oplus t'']$);

    $Set := Set \cup \{\mu'\}$;

    if $t_1 = 0$ *and* $(s_1, \nu_1) = (s_2, \nu_2)$ **then**

      $\mu'' = copy(\mu)$;

      trunc(μ'');

      μ''.append$([t_l \oplus t_4, (s'', \nu''))$;

      μ''.replace$([t', (i, s_1, \nu_1, t, s_3, \nu_3), t''], (t' \oplus t \oplus t''))$;

      μ''.`NumberOfHoles` $= \mu$.`NumberOfHoles` $- 1$;

      $Set := Set \cup \{\mu'\}$;

return Set**;**

F Witness Generation for TMPDA

In this section, we focus on the important question of generating a witness for an accepting run whenever our fixed-point algorithm guarantees non-emptiness. Since we use fixed-point computations to speed up our reachability algorithm, finding a witness, i.e., an explicit run witnessing reachability, becomes non-trivial. In fact, the difficulty of the witness generation depends on the system under consideration : while it is reasonably straight-forward for timed automata with no stacks, it is quite non-trivial when we have (multiple) stacks with non-well nested behavior.

Algorithm 13: Well-nested Timed Witness Generation

Function `WitnessTimedWR`($s_1, s_2, \nu, M = (\mathcal{S}, \Delta, s_0, \mathcal{S}_f,\ \mathcal{X}, n, \Sigma, \Gamma)$, *WRT*):
  Result: A sequence of transitions for an accepting run
  if $s_1 == s_2$ **then**
    return ϵ;
  if $\exists t = (s, \phi, R, \mathsf{nop}, s') \in \Delta \wedge \nu \models \phi \wedge \nu = R[\nu]$ **then**
    return t;
  forall $s', s'' \in \mathcal{S}$ **do**
    if $((s_1 \neq s') \vee (s'' \neq s_2)) \wedge (s', s'') \in$ *WRT*
    $\wedge \exists t = (s_1, \phi, R, \downarrow_i (\alpha), a, s') \in \Delta \wedge$
    $\exists t_2 = (s'', \phi', R', \uparrow_i (\alpha), a', s_2) \in \Delta \wedge \nu = R[\nu] = R[\nu'] \wedge \nu \models \phi \wedge \nu \models \phi'$
    then
      path=`WitnessTimedWR`(s', s'', ν, M, *WRT*);
      return $t.path.t_2$;
  forall $s \in M.\mathcal{S}$ **do**
    if $(s \neq s_1 \vee s \neq s_2) \wedge (s, 0, s_1) \in$ *WRT* $\wedge\, (s, 0, s_2) \in$ *WRT* **then**
      path1=`WitnessTimedWR`(s_1, s, ν, M, *WRT*);
      path2 = `WitnessTimedWR`(s, s_2, ν, M, *WRT*);
      return *path1.path2*;

0-holes. We start discussing the witness generation in the case of timed automata. As described in the algorithm in section 3, non-emptiness is guaranteed if a final state (s_f, ν_f) is reached from the initial state (s_0, ν_0) by computing the transitive closure of the transitions. The transitive closure computation results in generating a tuple $(s_0, \nu_0, t, s_f, \nu_f) \in$ `WRT` (Algorithm 10), for some time $0 \leq t \in \mathbb{R}$. Notice however that, in the Algorithms 10, we do not keep track of the sequence of states that led to the final state, and this is why we need to reconstruct a witness. To generate a witness run, we consider a *normal form* for any run in the underlying timed automaton, and check for the existence of a witness in the normal form. A run is in the normal form if it is a sequence of *time-elapse*, *useful*, and *useless* transitions. Time-elapse transitions have already been explained earlier. A discrete transition $(s, \nu) \rightarrow (s', \nu')$ is *useful* if $\nu \neq \nu'$, that is, there is at least one clock x such that $\nu'(x) = 0$ and $\nu(x) \neq 0$. A discrete transition is *useless* if $\nu = \nu'$.

Algorithm 14: Timed Pushdown Automata Witness Generation

Function `Witness`$((s_1,\nu_1), t, (s_2,\nu_2), M = (\mathcal{S}, \Delta, s_0, \mathcal{S}_f,\ \mathcal{X}, n, \Sigma, \Gamma)$, `WRT`):
    Result: A sequence of transitions for an accepting run
    forall $t_1 \in [T]$ **do**
        midPath = `Witness`$((s_1, \nu_1 + t_1), t - t_1, (s_2, \nu_2), M,$ `WRT`) Progress Measure 1;
        if *midPath* $\neq \emptyset$ **then**
            return $t_1 \cdot$*midPath*;
    forall $\delta = (s'', \phi', R', \mathsf{nop}, a', s_2) \in M.\Delta$ **do**
        if $\delta.R'[\nu_1] \neq \nu_1$ *and* $\nu_1 \models \delta.\phi'$) **then**
            $s_3 = \delta.s_2$;
            $\nu_3 = \delta.R'[\nu_1]$;
            midPath2 = `Witness`$((s_3, \nu_3), t, (s_2, \nu_2), M,$ `WRT`) Progress Measure 2;
            if *midPath2* $\neq \emptyset$ **then**
                return $\delta \cdot$*midPath2*;
    forall $s \in M.\mathcal{S}$ **do**
        path = `WitnessTimedWR`$(s_1, s, \nu_1, M,$`WRT`) Progress Measure 3;
        if *path* $\neq \emptyset$ **then**
            midPath3 = `Witness`$((s, \nu_1), t, (s_2, \nu_2), M,$`WRT`);
            if *midPath3* $\neq \emptyset$ **then**
                return *path* $\cdot$ *midPath3*;

If a tuple $(s_0, \nu_0, t, s_f, \nu_f)$, $t \geq 0$ is generated by Algorithm 10, we know that the system is non-empty. Now, we describe an algorithm to generate the witness run for obtaining $(s_0, \nu_0, t, s_f, \nu_f)$, by associating a *lexicographic progress measure* while exploring runs starting from (s_0, v_0). Integral time elapses, useful transitions and useless transitions are the three entities constituting the progress measure, ordered lexicographically.

- First we check if it is possible to obtain a witness run of the form $(s_0, \nu_0) \overset{t_1}{\rightsquigarrow} (s, \nu) \overset{t_2}{\rightsquigarrow} (s_f, \nu_f)$, where $\overset{t}{\rightsquigarrow}$ denotes a sequence of transitions whose total time elapse is t. In case $t_1, t_2 > 0$, with $t_1 + t_2 = t$, we can recurse on obtaining witnesses to reach (s, ν) from (s_0, ν_0), and (s_f, ν_f) from (s, ν), with strictly smaller time elapses, guaranteeing progress to termination.
- In case $t_1 = 0$ or $t_2 = 0$, we move to the second component of our progress measure, namely useful transitions. Assume $t_2 = 0$. Then indeed, there is no time elapse in reaching (s_f, ν_f) from (s, ν), but only a sequence of discrete transitions. Let $\#_X(\nu)$ denote the number of non-zero entries in the valuation ν. To obtain the witness, we look at a maximal sequence of useful transitions from (s, ν) of the form $(s, \nu) \to (s_1, \nu_1) \to \ldots \to (s_k, \nu_k)$ such that $\#_X(\nu) > \#_X(\nu_1) > \cdots > \#_X(\nu_k)$, where $k \leq$ the number of clocks. When we reach some (s_i, ν_i) from where we cannot make a useful transition, we go for a useless transition. Since there is no time elapse, and no useful resets, the clock valuations do not change on discrete transitions. We are left with enumerating all the locations to check the reachability to s_f (or to some s_j, from where we can again have a maximal sequence of useful transitions). Indeed, if (s_f, ν_f) is reachable from (s, ν) with no time elapse, there is a path having at most $|\mathcal{X}|$ useful transitions, interleaved with a sequence of useless transitions.

Generation of witness for timed automata is given in Algorithm 14. Notice that when $\kappa = (s_0, v_0, 0, s_f, v_f)$, the progress measure is $m(\kappa) = \#_X(\nu_0) - \#_X(\nu_f)$.

If $m(\kappa) = 0$, then $\nu_0 = \nu_f$, and the path takes only useless transitions. In this case, we consider the graph with nodes as states (s, ν), and there is an edge from (s_1, ν_1) to (s_2, ν_2) if there is a transition (s_1, φ, R, s_2) such that $\nu_1 \models \varphi$ and $\nu_1[R] = \nu_1$, that is, for all $x \in R$, $\nu_1(x) = 0$. If $m(\kappa) \neq 0$, then we take at least one useful transition. We can check if there exists a transition (s_1, φ, R, s_2) such that s_1 is reachable from s_0, and $\nu_0 \models \varphi, \nu_0[R] \neq \nu_0$, and the tuple $\kappa' = (s_2, \nu_0[R], 0, s_f, \nu_f) \in \texttt{WRT}$. In this case, we have $m(\kappa') < m(\kappa)$ and we can conclude by induction.

The case of a timed pushdown system with a single stack is similar to the case of timed automata, except for the fact that a discrete transition may involve push/pop operations. We use the same progress measures as in the timed automaton case, using the notion of runs in normal form.

Getting Witness from Holes. We can extend the backtracking algorithm for witness generation for MPDA to generate witness for TMPDA without much modification. In timed settings we need to take care of the time elapses within a hole and an atomic hole segment. When a hole is partitioned to an atomic hole segment and a hole, the time must be partitioned satisfying possible atomic hole segments and holes along with other constraints.