Be Lazy and Don't Care: Faster CTL Model Checking for Recursive State Machines*

Clemens Dubslaff[1], Patrick Wienhöft[1], and Ansgar Fehnker[2]

[1] Technische Universität Dresden, Dresden, Germany
{clemens.dubslaff,patrick.wienhoeft}@tu-dresden.de
[2] University of Twente, Enschede, The Netherlands
ansgar.fehnker@utwente.nl

Abstract. *Recursive state machines (RSMs)* are state-based models for procedural programs with wide-ranging applications in program verification and interprocedural analysis. Model-checking algorithms for RSMs and related formalisms and various temporal logic specifications have been intensively studied in the literature.
In this paper, we devise a new model-checking algorithm for RSMs and requirements in *computation tree logic (CTL)* that exploits the compositional structure of RSMs by ternary model checking in combination with a lazy evaluation scheme. Specifically, a procedural component is only analyzed in those cases in which it might influence the satisfaction of the CTL requirement. We evaluate our prototypical implementation on randomized scalability benchmarks and on an interprocedural data-flow analysis of Java programs, showing both practical applicability and significant speedups in comparison to state-of-the-art model-checking tools for procedural programs.

1 Introduction

Model checking [4,12] is a well-established technique for verifying that a system model meets a given requirement. System models are most commonly given as *Kripke structures*, i.e., directed graphs over states whose edges model the operational behavior of the system with labels over a set of atomic propositions specifying properties of states. Over these labels, requirements are usually formalized in a temporal logic such as *computation tree logic (CTL, [11])*.

In this paper, we revisit the model-checking problem for *recursive state machines (RSMs)* models and CTL requirements [1]. RSMs provide a standard model for the operational behavior of programs with recursive procedure calls. They closely follow the compositional structure of the procedural program by modeling procedures by separate Kripke structures (called *components*) that are connected through *call* and *return* nodes. Due to the infinite-state semantics

* The authors are supported by the DFG through the Cluster of Excellence EXC 2050/1 (CeTI, project ID 390696704, as part of Germany's Excellence Strategy) and the TRR 248 (see https://perspicuous-computing.science, project ID 389792660).

of RSMs, the standard CTL model-checking algorithm for finite Kripke structures [4,11] is not directly applicable [1]. Fortunately, the satisfaction of a given CTL formula in a component of an RSM solely depends on the satisfaction of subformulas in return nodes of the component, so-called *contexts* [9,3]. Intuitively, contexts model the environmental influence on the component, i.e., how the satisfaction of the formula depends on the calling component. Exhaustively generating all contexts that could arise during program execution and applying the standard CTL model-checking algorithm for finite Kripke structures on components directly leads to an algorithm to model check RSMs against CTL formulas [1]. This algorithm runs in exponential time in the size of the RSM due to possibly exponentially many contexts that have to be considered for each component. Since the model-checking problem for RSMs and CTL formulas is ExpTime-complete [5], this algorithm cannot be improved in the worst case. Nevertheless, there is plenty of room for heuristic optimizations that might show runtime improvements in practice.

This paper devises a new method to reduce the number of subformulas and contexts evaluated during the model-checking decision procedure, following a lazy rather than an exhaustive deduction scheme. The main idea behind our *lazy approach* is to use *ternary model checking* and successively refine the global satisfaction relation by step-wise evaluating new contexts that could contribute to deciding the overall model-checking problem [14]. While our lazy approach might also have to consider all subformulas and contexts in the worst case, this is usually not the case in practice, as we show in this paper.

We implemented a ternary variant of the exhaustive approach by Alur et al. [1] and our new lazy approach in a tool called RSMCheck[3]. To the best of our knowledge, RSMCheck is the first model checker specifically dedicated to RSMs, while existing state-of-the-art model checkers for procedural programs such as PDSolver [15] and PuMoC [20] rely on pushdown systems. RSMs and pushdown systems can be linearly transformed to each other while preserving their Kripke structure semantics (see, e.g., [6]). However, RSMs have the advantage of directly reflecting the compositional structure of a procedural program and providing an intuitive visual representation. To this end, choosing RSMs as model for procedural programs can ease the interpretation of counterexamples and witnesses generated by model checking and hence facilitate debugging during program development steps.

We conduct three experimental studies for RSMCheck, addressing scalability, comparison to existing model-checking tools, and application to real-world examples in terms of an interprocedural data-flow analysis on Java programs. In these studies we show that our lazy approach is effective, evaluates less contexts than in the exhaustive case, and leads to significant speedups up to one order of magnitude compared to the exhaustive approach. Applied on their own benchmark suites, PDSolver and PuMoC show timeouts or exceed memory constraints on several instances [20]. We demonstrate that our lazy approach

[3] The tool along with data to reproduce our experimental studies can be downloaded at `https://github.com/PattuX/RSMCheck`.

manages to verify all instances and outperforms PDSolver and PuMoC by being up to two orders of magnitude faster.

Outline. After settling notations and basic definitions required to formally state the CTL model-checking problem for RSMs in Section 2, we first extend the exhaustive model-checking approach by Alur et al. [1] to the ternary setting in Section 3. The lazy approach is detailed in Section 4 and evaluated in Section 5. We close the paper with further related work and future work in Section 6.

2 Preliminaries

For a set X we denote by $\wp(X)$ the power set of X and by X^*, X^+, and X^ω the sets of finite, finite non-empty, and infinite sequences of elements in X, respectively. Given a sequence $\pi = x_1, x_2, \ldots$, we denote by $\pi[i] = x_i$ the ith element of π. A *(ternary) interpretation* over X is a function $\partial\colon X \to \{\mathtt{tt}, \mathtt{ff}, \mathtt{??}\}$ where $\mathtt{tt}$ stands for "true", $\mathtt{ff}$ for "false", and $\mathtt{??}$ for "unknown". We denote by $\Delta(X)$ the set of all interpretations over X. An interpretation $\partial \in \Delta(X)$ is a *refinement* of $\partial' \in \Delta(X)$ if for all $x \in X$ we have $\partial'(x) = \mathtt{tt}$ implies $\partial(x) = \mathtt{tt}$, and $\partial'(x) = \mathtt{ff}$ implies $\partial(x) = \mathtt{ff}$.

A *Kripke structure* (see, e.g., [4]) is a tuple $\mathcal{K} = (S, \longrightarrow, AP, L)$ where S is a set of states, $\longrightarrow \subseteq S \times S$ is a transition relation, AP is a finite set of atomic propositions, and $L\colon S \to \wp(AP)$ is a labeling function that labels states with atomic propositions. To ease notations, we write $s \longrightarrow s'$ for $(s, s') \in \longrightarrow$. A *path* in $\mathcal{K}$ is a sequence $s_1, s_2, \ldots \in S^\omega$ where for each $i \in \mathbb{N}$ we have $s_i \longrightarrow s_{i+1}$. The set of all paths starting in a state $s \in S$ is denoted by $\Pi(s)$.

2.1 Computation Tree Logic

To reason about Kripke structures we specify system requirements in *computation tree logic* (CTL, [11]). A CTL formula over AP is defined by the grammar

$$\Phi \;=\; \mathtt{tt} \mid a \mid \neg\Phi \mid \Phi \vee \Phi \mid \exists \mathsf{X}\Phi \mid \exists \mathsf{G}\Phi \mid \exists \Phi \,\mathsf{U}\, \Phi$$

where a ranges over AP. Further standard operators, e.g., $\wedge$, F, and $\forall$, can be derived through standard transformations such as DeMorgan's rule [4]. We denote by $\mathit{Subf}(\Phi)$ and $\mathit{Subf}_\exists(\Phi)$ the set of subformulas and existential quantified subformulas of Φ, respectively. Given a Kripke structure $\mathcal{K} = (S, \longrightarrow, AP, L)$ we define the satisfaction relation $\models$ for CTL formulas over AP recursively by

$$\begin{array}{llll}
s \models \mathtt{tt} & & s \models \Phi_1 \vee \Phi_2 & \text{iff } s \models \Phi_1 \text{ or } s \models \Phi_2 \\
s \models a & \text{iff } a \in L(s) & s \models \exists \mathsf{X}\Phi & \text{iff } \exists \pi \in \Pi(s).\pi[2] \models \Phi \\
s \models \neg\Phi & \text{iff } s \not\models \Phi & s \models \exists \mathsf{G}\Phi & \text{iff } \exists \pi \in \Pi(s).\forall i \in \mathbb{N}.\pi[i] \models \Phi \\
\multicolumn{4}{l}{s \models \exists \Phi_1 \,\mathsf{U}\, \Phi_2 \;\text{ iff }\; \exists \pi \in \Pi(s), j \in \mathbb{N}.\forall i < j.\pi[i] \models \Phi_1 \wedge \pi[j] \models \Phi_2}
\end{array}$$

An interpretation ∂ over $S \times \mathit{Subf}(\Phi)$ is *consistent* with $\mathcal{K}$ if for all $s \in S$ and $\phi \in \mathit{Subf}(\Phi)$ we have $\partial(s, \phi) = \mathtt{tt}$ implies $s \models \phi$ and $\partial(s, \phi) = \mathtt{ff}$ implies $s \not\models \phi$.

2.2 Recursive State Machines

A labeled *recursive state machine* (RSM, [1]) over a set of atomic propositions AP is a tuple $\underline{\mathcal{A}} = (\mathcal{A}_1, \ldots, \mathcal{A}_k)$ comprising *components*

$$\mathcal{A}_i \;=\; (N_i, B_i, Y_i, En_i, Ex_i, \longrightarrow_i, I_i, AP, L_i)$$

for $i = 1, \ldots, k$ where

- N_i is a set of nodes for which $N_i \cap N_j = \varnothing$ for all $j = 1, \ldots, k$, $i \neq j$,
- B_i is a set of boxes for which $B_i \cap B_j = \varnothing$ for all $j = 1, \ldots, k$, $i \neq j$,
- $Y_i \colon B_i \to \{1, \ldots, k\}$ is a mapping assigning a component index to every box,
- $En_i, Ex_i \subseteq N_i$ with $En_i \cap Ex_i = \varnothing$, are sets of *entry* and *exit nodes*, respectively,
- $\longrightarrow_i \subseteq (N_i \setminus Ex_i) \cup \mathit{Return}_i \times (N_i \setminus En_i) \cup \mathit{Call}_i$ is a transition relation, and
- $L_i \colon N_i \cup \mathit{Call}_i \cup \mathit{Return}_i \to \wp(AP)$ is a node labeling function for which $L_i((b,n)) = L_{Y(b)}(n)$ for all $(b,n) \in \mathit{Call}_i \cup \mathit{Return}_i$.

Here, $\mathit{Call}_i = \bigcup_{b \in B_i} \mathit{Call}_b$ where $\mathit{Call}_b = \{(b, en) \mid en \in En_{Y(b)}\}$ denotes the set of *call nodes* of a box b and $\mathit{Return}_i = \bigcup_{b \in B_i} \mathit{Return}_b$ where $\mathit{Return}_b = \{(b, ex) \mid ex \in Ex_{Y(b)}\}$ denotes the set of *return nodes* of a box b. We assume that all nodes except exit nodes are not final, i.e., for all $i \in \{1, \ldots, k\}$ and $n \in (N_i \setminus Ex_i) \cup \mathit{Return}_i$ there is $n' \in (N_i \setminus En_i) \cup \mathit{Call}_i$ such that $n \longrightarrow_i n'$. Note that we allow for direct transitions from return to call nodes. By omitting component indices, we denote the union of all corresponding entities in the RSM, e.g., we write N for $\bigcup_{i=1}^{k} N_i$, B for $\bigcup_{i=1}^{k} B_i$, and $\longrightarrow$ for $\bigcup_{i=1}^{k} \longrightarrow_i$.
The semantics of a component $\mathcal{A}_i$ is defined as Kripke structure $[\![\mathcal{A}_i]\!] = (N_i \cup \mathit{Call}_i \cup \mathit{Return}_i, \longrightarrow_i, AP, L_i)$. The semantics of $\underline{\mathcal{A}}$ is a Kripke structure

$$[\![\underline{\mathcal{A}}]\!] \;=\; (B^* \times (N \cup \mathit{Call} \cup \mathit{Return}), \Longrightarrow, AP, L)$$

where L labels each state as the corresponding node, i.e., $L((\sigma, n)) = L_i(n)$ for all $\sigma \in B^*$ and $n \in N_i \cup \mathit{Call}_i \cup \mathit{Return}_i$, and $\Longrightarrow$ is the smallest transition relation that obeys the following rules:

$$\text{(loc)}\ \frac{\sigma \in B^* \qquad n \longrightarrow n'}{(\sigma, n) \Longrightarrow (\sigma, n')} \qquad \text{(call)}\ \frac{\sigma b \in B^+ \qquad (b, en) \in \mathit{Call}_b \qquad en \longrightarrow n}{\big(\sigma, (b, en)\big) \Longrightarrow (\sigma b, n)}$$

$$\text{(loop)}\ \frac{ex \in Ex}{(\varepsilon, ex) \Longrightarrow (\varepsilon, ex)} \qquad \text{(return)}\ \frac{\sigma b \in B^+ \qquad (b, ex) \in \mathit{Return}_b \qquad (b, ex) \longrightarrow n}{(\sigma b, ex) \Longrightarrow (\sigma, n)}$$

Intuitively, a state (σ, n) of the Kripke structure $[\![\underline{\mathcal{A}}]\!]$ comprises a *call stack* σ and a local node n of some component of $\underline{\mathcal{A}}$. Rule (loc) represents an internal transition of a component, (loop) implements that the execution stays in the exit nodes when leaving the outermost component, and (call) and (return) formalize entering and leaving a box, respectively. For a CTL formula Φ, we write $\underline{\mathcal{A}} \models \Phi$ if for all $n \in En_1$ we have $(\varepsilon, n) \models \Phi$ in $[\![\underline{\mathcal{A}}]\!]$ [9,10]. The *model-checking problem* we consider here in this paper asks whether $\underline{\mathcal{A}} \models \Phi$ for a given RSM $\underline{\mathcal{A}}$ and CTL formula Φ, both over AP.

```
1  public class Dataflow {
2    public static int i;
3    public static void main(String[]
         ↪ args) {
4      i = 5 / 7; a(); b();
5    }
6    public static void a() {
7      // do nothing
8    }
9    public static void b() {
10     i = i * i;
11   }
12 }
```

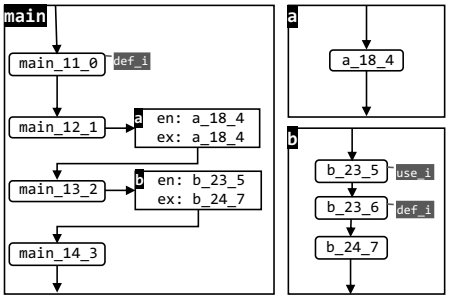

Fig. 1: Java Dataflow example from [15] and its generated control-flow RSM.

Example. Figure 1 depicts a Java program (left) and an automatically generated RSM model (right). Nodes in the RSM stand for control-flow locations with names encoding references back to the abstract syntax tree of the source code. Furthermore, nodes are labeled with use_i and def_i, which indicate whether the variable i is read or written, respectively. Model checking on RSMs with such use-def annotations can be used for an interprocedural data-flow analysis. For instance, the requirement that whenever the variable i is defined, it is eventually used, can be expressed by the CTL formula $\forall\mathsf{G}\big(\mathsf{def}_i \to \exists\mathsf{F}(\mathsf{use}_i)\big)$. Our Dataflow example does not meet this requirement: after squaring i in Line 10, the new value of i is not used in later program execution steps. In the RSM of Figure 1, this is witnessed by the only existing execution that starts in the initial node main_11_0, reaches the def_i-labeled node b_23_6 after calling b(), and finally continues with b_24_7 and main_14_3 that are both not labeled with use_i.

3 Ternary RSM Model Checking

This section provides the foundations for our exhaustive and lazy model-checking algorithms. For this, we closely follow the approach of [1] and adapt their algorithm for model checking single-exit RSMs against CTL* requirements towards a ternary model-checking algorithm of multi-exit RSMs against CTL. Multi-exit RSMs, i.e., RSMs where components might have more than one exit node, are especially relevant for modeling real-world procedural programs. In fact, except the Dataflow example from Figure 1, all examples we consider in our experimental studies of Section 5 require multi-exit RSMs for their analysis. Meanwhile, CTL as a subclass of CTL* is still expressive enough to specify lots of relevant properties, e.g., use-def properties for interprocedural static analysis.

The support of ternary CTL model checking follows the ideas by [7] and replaces the role of refinement operations on satisfaction sets as employed in [1]. To ensure compositional RSM model checking, we discuss two kinds of deductions: first, how ternary interpretations are refined locally on each component, and second, how ternary refinements are globally propagated.

Algorithm 1: ContextualizE($\underline{\mathcal{A}}, \Phi, \underline{\partial}, b$)

input : an RSM $\underline{\mathcal{A}} = (\mathcal{A}_1, \dots, \mathcal{A}_k)$, a CTL formula Φ, a vector $\underline{\partial} = (\partial_1, \dots, \partial_k)$ of ternary interpretations ∂_j for $\mathcal{A}_j$, and a box $b \in B_i$

output: a modified RSM $\underline{\mathcal{A}}'$ with a Φ-contextualized interpretation $\underline{\partial}'$

$\gamma_b := \{(ex, \phi, \eta) \mid \phi \in \mathit{Subf}(\Phi), (b, ex) \in \mathit{Return}_b, \partial_i((b, ex), \phi) = \eta\}$
if *there is* j *where* $\gamma_b \subseteq \partial_j$ **then**
  $\underline{\mathcal{A}}' := \underline{\mathcal{A}}$
  $Y_i'(b) := j$
else
  $\mathcal{A}_{k+1} := \mathcal{A}_{Y_i(b)}$
  $\underline{\mathcal{A}}' := (\mathcal{A}_1, \dots, \mathcal{A}_k, \mathcal{A}_{k+1})$
  $\partial_{k+1} := \partial_{Y_i(b)}$
  forall $(s, \phi, \eta) \in \gamma_b$ **do** $\partial_{k+1}(s, \phi) := \gamma_b(s, \phi)$
  $\underline{\partial}' := (\partial_1, \dots, \partial_k, \partial_{k+1})$
  $Y_i'(b) := k + 1$
return $\underline{\mathcal{A}}', \underline{\partial}'$

3.1 Local Deduction

To locally refine ternary interpretations on RSM components, we use a function LocalDeduce($\mathcal{K}, \Phi, \partial$) that maps a finite Kripke structure $\mathcal{K} = (S, \longrightarrow, AP, L)$, a CTL formula Φ over AP, and an interpretation $\partial\colon S \times \mathit{Subf}(\Phi) \to \{\mathtt{tt}, \mathtt{ff}, \mathtt{??}\}$ that is consistent with $\mathcal{K}$ to an interpretation $\partial'\colon S \times \mathit{Subf}(\Phi) \to \{\mathtt{tt}, \mathtt{ff}, \mathtt{??}\}$ refining ∂. In essence, LocalDeduce implements one step of the CTL model-checking algorithm by [7] where interpretations on subformulas are refined in a bottom-up fashion as in classical CTL model checking [11] but on ternary interpretations instead of binary ones. To achieve ternary deduction, an optimistic and a pessimistic run of the classical CTL deduction step is performed on binary interpretations of subformulas. In the optimistic run all subformulas that are "unknown" are assumed to hold, while in the pessimistic run they are assumed to not hold. Then, all subformulas that do not hold after the optimistic run do surely not hold in the ternary setting and likewise, all subformulas that do hold after the pessimistic run surely hold.

3.2 Contextualization of Components

A slight difference of our LocalDeduce method compared to a single deduction step by [7] is that we explicitly give an arbitrary consistent partial interpretation ∂ as input parameter, while the algorithm by [7] assumes a maximally refined consistent partial interpretation over all subformulas. To this end, we can include assumptions on the satisfaction of subformulas in the deduction process such as knowledge on the environment the system is executed in. In the setting of RSMs, the environment of a component is constituted by their calling components. Specifically, following the notion of *contexts* [3,9], the environmental influence on

Algorithm 2: GlobalDeduce($\underline{\mathcal{A}}, \Phi, \underline{\partial}$)

input : an RSM $\underline{\mathcal{A}} = (\mathcal{A}_1, \ldots, \mathcal{A}_k)$, a CTL formula Φ, and a vector $\underline{\partial} = (\partial_1, \ldots, \partial_k)$ of ternary interpretations ∂_j for $\mathcal{A}_j$
output: refined interpretations $\underline{\partial}'$ of $\underline{\partial}$

```
1 ∂' := ∂
2 repeat
3 |   ∂̂ := ∂'
4 |   forall i ∈ {1, ..., k} do
5 |   |   forall (b, en) ∈ Call_i do
6 |   |   |   ∂'_i((b, en), Φ) := ∂'_{Y_i(b)}(en, Φ)
7 |   |   ∂'_i := LocalDeduce([[A_i]], Φ, ∂'_i)
8 until ∂̂ = ∂'
9 return ∂'
```

a component can be fully captured by a given satisfaction relation on existential formulas in exit nodes of the components.[4] For an RSM $\underline{\mathcal{A}}$ and a CTL formula Φ both over a set of atomic propositions AP as formalized in Section 2, a *Φ-context* of a component $\mathcal{A}_i$ in $\underline{\mathcal{A}}$ is formalized as an interpretation $\gamma_i \in \Delta\big(Ex_i \times Subf_{\exists}(\Phi)\big)$ over the component's exit nodes and existential subformulas of Φ.

To reason about components in a modular way, we have to keep track of the contexts and deduction results under these contexts for their reuse. This is achieved by the function Contextualize, described in Algorithm 1, which maps $\underline{\mathcal{A}}$, a tuple $\underline{\partial} = (\partial_1, \ldots, \partial_k)$ of local interpretations for components $\mathcal{A}_1, \ldots, \mathcal{A}_k$ of $\underline{\mathcal{A}}$, and a target box $b \in B_i$ to a possibly modified RSM $\underline{\mathcal{A}}'$ with a Φ-contextualized interpretation $\underline{\partial}'$. Our algorithm for Contextualize checks whether we already considered the component assigned to b w.r.t. the context induced from b's return nodes. If this is the case, we (re)assign b to the found contextualized component. Otherwise a copy[5] $\mathcal{A}_{k+1}$ of the component $\mathcal{A}_{Y_i(b)}$ with the new context is generated (i.e., the number of components of the RSM increases from k to $k+1$) and the box b is reassigned to the fresh component $\mathcal{A}_{k+1}$ by updating function Y_i (see Section 2.2).

3.3 Global Deduction

To propagate information from inside a component to a calling component, we use a function GlobalDeduce, described in Algorithm 2, that maps an RSM $\underline{\mathcal{A}}$, a target CTL formula Φ, and a tuple $\underline{\partial} = (\partial_1, \ldots, \partial_k)$ of local interpretations to refined interpretations $\underline{\partial}' = (\partial'_1, \ldots, \partial'_k)$. Our algorithm for GlobalDeduce starts with $\underline{\partial}' = \underline{\partial}$ and performs the following two steps until a fixed point is

[4] Since the standard CTL model-checking deduction follows a backward-search approach, the contextual information contained in the exit nodes of the component propagates towards the entry nodes of the component during a local deduction step.

[5] This is done due to better understandability of the approach. For practical implementations, one might only copy and modify interpretations on the components.

Algorithm 3: ExhaustiveCheck($\underline{\mathcal{B}}, \Phi$)

input : an RSM $\underline{\mathcal{B}} = (\mathcal{B}_1, \ldots, \mathcal{B}_\ell)$ and a CTL formula Φ, both over AP
output: tt if $\underline{\mathcal{B}} \models \Phi$ and ff if $\underline{\mathcal{B}} \not\models \Phi$

1 $\underline{\mathcal{A}}, \underline{\partial} :=$ Initialize($\underline{\mathcal{B}}, \Phi$)
2 $F := \varnothing$
3 **while** $F \neq \mathit{Subf}(\Phi)$ **do**
4 Pick $\phi \in \mathit{Subf}(\Phi)$ with $\mathit{Subf}(\phi) \setminus F = \{\phi\}$
5 $F := F \cup \{\phi\}$
6 **repeat**
7 **forall** $b \in B$ **do** $\underline{\mathcal{A}}, \underline{\partial} :=$ Contextualize($\underline{\mathcal{A}}, \phi, \underline{\partial}, b$)
8 $\underline{\partial} :=$ GlobalDeduce($\underline{\mathcal{A}}, \phi, \underline{\partial}$)
9 **until** $\underline{\partial}$ *did not change*
10 **forall** $i \in \{1, \ldots, k\}$, $n \in N_i$ *with* $\partial_i(n, \phi) =$?? **do**
11 **if** $\phi = \exists \mathsf{G} \psi$ **then** $\partial_i(n, \phi) :=$ tt
12 **if** $\phi = \exists \psi_1 \,\mathsf{U}\, \psi_2$ **then** $\partial_i(n, \phi) :=$ ff
13 **if** *there is* $en \in En_1$ *with* $\partial_1(en, \Phi) =$ ff **then return** ff
14 **else return** tt

reached for the local interpretations, i.e., $\underline{\partial}'$ does not change anymore: First, a local deduction step LocalDeduce($\mathcal{A}_i, \Phi, \partial_i$) is performed for each component $\mathcal{A}_i$ and their current interpretations ∂_i. Second, we copy the refined interpretations on the entry nodes of each component $\mathcal{A}_i$ to their corresponding call nodes in the calling component. This refinement in the call nodes may cause new possible local deductions in the calling components, leading to further refinements in their entry nodes. As such, we alternate between these two steps until we reach a fixed point.

3.4 Exhaustive Approach to RSM Model Checking

Piecing together the algorithms sketched so far, we define a compositional algorithm for model checking RSMs against CTL formulas. That is, the algorithm runs locally on the components of the RSM and propagates their satisfaction relations towards a global satisfaction relation. The procedure follows ideas from [1] where satisfaction of CTL subformulas is evaluated in a bottom-up fashion, determining the truth value of minimal subformulas in all nodes before proceeding to larger subformulas. During the evaluation, contextualized components are created whenever there is not enough information present to fully determine the truth values for subformulas in all nodes of calling components. Algorithm 3 shows the decision procedure ExhaustiveCheck($\underline{\mathcal{A}}, \Phi$) that decides for an RSM $\underline{\mathcal{A}}$ and a CTL formula Φ whether $\underline{\mathcal{A}} \models \Phi$ or not. The algorithm starts with an initialization of the local ternary interpretations of the components of $\underline{\mathcal{A}}$ (function Initialize, see in Line 1). Specifically, Initialize sets all local interpretations to evaluate to ?? and then performs a local deduction for $\mathcal{A}_1$ to determine basic truth assignments in the exit nodes of $\mathcal{A}_1$ following rule (loop) in the definition

of RSM semantics. After initialization, ExhaustiveCheck iterates over all subformulas of Φ in a bottom-up fashion as within classical CTL model checking. For each formula we alternate between contextualizing components assigned to boxes by Contextualize and a global deduction by GlobalDeduce, refining local interpretations of components and determining new contexts towards a propagation from calling components to called ones. This is done until we reach a fixed point, i.e., local interpretations are not refined any further by GlobalDeduce.

Global dependency cycle resolution. The reached fixed point does not solely ensure that all truth values for the considered subformula are determined in all nodes, i.e., some local interpretations may still map to ??. This can happen when the context of a box depends on the evaluation of the boxes' entry nodes. To illustrate this situation, let us consider an example RSM $\underline{\mathcal{A}} = (\mathcal{A}_1, \mathcal{A}_2)$ over $AP = \{\bigcirc, \bullet, \bullet\}$ depicted in Figure 2: The truth value of $\Phi = \exists \mathsf{X} \exists \mathsf{G} \bigcirc$ in n_1 depends on the truth value of $\phi = \exists \mathsf{G} \bigcirc$ in the return node (b, n_7), providing the context of $\mathcal{A}_2$ in its exit node n_7. However, we cannot deduce this truth value locally in $\mathcal{A}_1$ as it depends on whether ϕ holds in the call node (b, n_6) or not. Intuitively, we thus have a cycle of dependencies connected through several components that hinders further refinement via Contextualize and GlobalDeduce. We resolve such situations by the following reasoning: Since there is a dependency cycle that hindered refinement, all nodes on this cycle have to satisfy $\bigcirc$. Thus, this cycle can serve as a witness of ϕ to hold and we refine all local interpretations for ϕ and nodes on the cycle towards tt. A similar argumentation can be applied when ϕ is an until formula but with refining all ??-nodes towards ff. For instance, $\bullet$ is not reachable from (b, n_6), such that $\phi = \exists \bigcirc \mathsf{U} \bullet$ cannot hold on the dependency cycle illustrated above. Note that our efficient resolution of global dependency cycles relies on ternary deduction, since cycles of ??-nodes directly provide information about undeducibility of truth values. While our algorithm is based on [1], their algorithm uses binary refinements and thus cannot exploit such a resolution. However, their algorithm also includes mechanisms to reason about satisfaction of formulas expressed in linear temporal logic (LTL), which is used to cover the cycle resolution step.

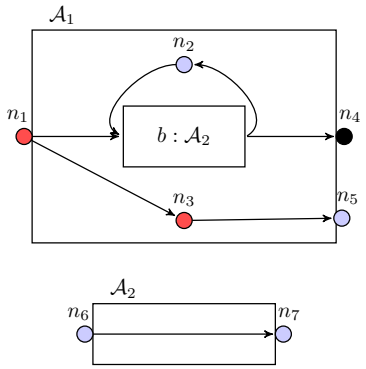

Fig. 2: Example RSM

Exhaustive RSM Model Checking. Taking global dependency cycle resolution into account and with proof techniques from [7,1], we obtain correctness of our exhaustive model-checking algorithm ExhaustiveCheck:

Theorem 1. ExhaustiveCheck$(\underline{\mathcal{A}}, \Phi)$ *terminates for any RSM* $\underline{\mathcal{A}}$ *and CTL formula* Φ *over a common set of atomic propositions and returns* tt *iff* $\underline{\mathcal{A}} \models \Phi$.

Proof sketch. We show termination and soundness of each of the subroutines Initialize, LocalDeduce, GlobalDeduce and Contextualize, and then

lift the results to the full algorithm for ExhaustiveCheck. First, observe that Initialize can be seen of a special case of a Contextualize where the context is given by the rule (loop) in the definition of $[\![\underline{\mathcal{A}}]\!]$.

The termination of LocalDeduce directly follows from [7]. Termination of Contextualize is straight forward (see Algorithm 1). For GlobalDeduce the important observation is that it strictly refines an interpretation until a fixed point is reached, which is done in finitely many steps as the set of nodes N and subformulas $\mathit{Subf}(\Phi)$ are finite. Since the number of contextualizations of each box is bounded by $3^{|Ex|\cdot|\mathit{Subf}_\exists(\Phi)|}$, the calls of Contextualize in Line 7 can only add finitely many contextualized components to $\underline{\mathcal{A}}$. Further, GlobalDeduce is idempotent and $\underline{\partial}$ does not change if $\underline{\mathcal{A}}$ did not change. Thus, each iteration of the main loop from Line 3 to Line 12 is guaranteed to terminate. Lastly, it is clear that the main loop is executed exactly once for each $\phi \in \mathit{Subf}(\Phi)$ and thus the algorithm terminates.

For soundness, we show that at each execution point of the algorithm the computed partial interpretation $\underline{\partial}$ is sound, i.e.,

$$\underline{\partial}(s,\phi) = \mathtt{tt} \implies \forall \sigma \in B^* : (\sigma, s) \models \phi$$

and

$$\underline{\partial}(s,\phi) = \mathtt{ff} \implies \forall \sigma \in B^* : (\sigma, s) \not\models \phi.$$

Soundness of LocalDeduce follows immediately from [7]. For GlobalDeduce and Contextualize the statement follows from using the definition of the underlying Kripke structure $[\![\underline{\mathcal{A}}]\!]$ of $\underline{\mathcal{A}}$ and the soundness of LocalDeduce. The main effort in the proof is to show that the assertions following Line 10 are correct. The arguments here follow the same ideas as outlined in the last section about global dependency cycle resolution. □

4 Lazy RSM Model Checking

The model-checking algorithm presented in Section 3 mainly combined existing techniques for model-checking RSMs and CTL formulas [11,9,7,3]. In this section, we devise a new algorithm that uses elements of the former but aims towards reducing the number of deduction steps involved. This is achieved by exploiting the structure of the target CTL formula and the compositional structure of the RSM towards lazy evaluation of subformulas and components, respectively.

4.1 Lazy Contextualization

Exhaustive RSM model checking determines satisfaction of subformulas $\phi \in \mathit{Subf}(\Phi)$ in all nodes of the RSM $\underline{\mathcal{A}}$ by evaluating the satisfaction relation within components w.r.t. all possible contexts. The possibly exponentially many contexts that have to be considered with this approach is the main reason for CTL model checking over RSMs to be EXPTIME-complete [5]. Reducing the number

of contexts considered during the deduction process thus provides a potential to speed up the model checking of RSMs.

Ternary formula evaluation. The main idea towards reducing the number of contexts to be evaluated is to leave satisfaction of subformulas ϕ of Φ unspecified in case they do not have any influence on the satisfaction of Φ. For instance, let us consider the RSM of Figure 2 and $\Phi = \exists \mathsf{X} \bigcirc \vee \exists \mathsf{X}(\exists \bigcirc \mathsf{U} \bullet)$. Then, satisfaction of Φ can be determined by solely regarding $\phi = \exists \mathsf{X} \bigcirc$ in n_1 and not reasoning about either disjunct in other nodes, which would be necessarily done in the bottom-up approach. Further, evaluating ϕ in n_1 does not require any contextualization of box b since n_3 is labeled by $\bigcirc$ and thus, in component $\mathcal{A}_1$ we can already locally deduce ϕ to hold in n_1 and thus $n_1 \models \Phi$, directly leading to $\mathcal{A} \models \Phi$. In this example, we reduced the number of contexts to be evaluated as we did not evaluate any context for component $\mathcal{A}_2$.

Lazy expansion. To determine those contexts that have to be evaluated to solve the model-checking problem, we combine the ternary formula evaluation with a heuristic that determines those contexts that might be the reason for underspecified satisfaction of subformulas and impact satisfaction of Φ in the RSM. We provide such a heuristic by the function GetNextExpansion, specified by Algorithm 4. Depending on a node n where it is unknown whether the target formula Φ holds or not, this function selects a box for which a contextualization step in combination with a global deduction (see Section 3.3) could determine the truth value of Φ in n. GetNextExpansion is defined in a recursive manner, traversing Φ in a top-down fashion to reason on *why* Φ is unknown in n and to find a box b where adding a subformula to its context might refine the interpretation of Φ in n. By lazily contextualizing heuristically selected boxes rather than contextualizing all boxes as in the case of the exhaustive approach, we can potentially save contextualization steps.

Algorithm 4 considers several cases during recursion, from which we exemplify the most significant ones. First, those properties that could be locally resolved are considered. For instance, Line 2 deals with Φ being a disjunction where it is known that at least one disjunct must be unknown since otherwise Φ would be determined in n. Then, a disjunct ϕ_i is chosen nondeterministically and GetNextExpansion is recursively called, determining which contextualization could resolve whether ϕ_i holds in n. The cases of entering and leaving a box b are considered in Line 5 and Line 6, respectively. Notably, if n is an exit node, we consider the satisfaction of Φ in the calling component, i.e., in its return node. If Φ is already known, we found a box where contextualizing yields additional information and thus return that box as our base case in Line 8. Otherwise, we continue our search. For existential path properties, let us exemplify the case where $\Phi = \exists \phi \, \mathsf{U} \, \psi$ (see Line 10). Here, we determine the next recursive call arguments following the well-known CTL expansion law $\Phi = \psi \vee \big(\phi \wedge \exists \mathsf{X}(\phi \, \mathsf{U} \, \psi)\big)$. First, we consider the local cases where ψ or ϕ are unknown in n, asking for a box to contextualize by invoking GetNextExpansion on ψ and ϕ, respectively. Otherwise, the reason for Φ being unknown in n cannot be locally given and we continue in a successor node of n where Φ is still unknown.

Algorithm 4: GetNextExpansion$(\underline{\mathcal{A}}, n, \Phi, \underline{\partial}, \sigma)$

input : RSM $\underline{\mathcal{A}} = (\mathcal{A}_1, \ldots, \mathcal{A}_k)$, node $n \in N_i$, formula Φ, a vector $\underline{\partial} = (\partial_1, \ldots, \partial_k)$ of interpretations ∂_j for $\mathcal{A}_j$, and a call stack $\sigma \in B^+$
output: a box b to contextualize

1 // ... other local case $\Phi = \neg\phi$ ...
2 **if** $\Phi = \phi_1 \vee \ldots \vee \phi_\ell$ **then**
3 | choose $j \in \{1, \ldots, \ell\}$ with $\partial_i(n, \phi_j) = ??$
4 | **return** GetNextExpansion$(\underline{\mathcal{A}}, n, \phi_j, \underline{\partial}, \sigma)$
5 **if** $n = (b, en) \in Call_i$ **then return** GetNextExpansion$(\underline{\mathcal{A}}, en, \Phi, \underline{\partial}, \sigma b)$
6 **if** $n \in Ex_i$ *and there are* $\rho \in B^*$ *and* $b \in B$ *with* $\rho b = \sigma$ **then**
7 | **if** $\partial_{Y_i(b)}((b, n), \Phi) = ??$ **then return** GetNextExpansion$(\underline{\mathcal{A}}, (b, n), \Phi, \underline{\partial}, \rho)$
8 | **return** b // base case
9 // ... other existential cases $\Phi = \exists \mathsf{G}\phi$ and $\Phi = \exists \mathsf{X}\phi$ ...
10 **if** $\Phi = \exists \phi \,\mathsf{U}\, \psi$ **then**
11 | **if** $\partial_i(n, \psi) = ??$ **then return** GetNextExpansion$(\underline{\mathcal{A}}, n, \psi, \underline{\partial}, \sigma)$
12 | **if** $\partial_i(n, \phi) = ??$ **then return** GetNextExpansion$(\underline{\mathcal{A}}, n, \phi, \underline{\partial}, \sigma)$
13 | choose n' with $n \longrightarrow_i n'$ and $\partial_i(n', \Phi) = ??$
14 | **return** GetNextExpansion$(\underline{\mathcal{A}}, n', \Phi, \underline{\partial}, \sigma)$

Global dependency cycle resolution. Similar as in the case of exhaustive RSM model checking (see Section 3.4), global dependency cycles are an issue also within GetNextExpansion. When implementing GetNextExpansion exactly as described in Algorithm 4, the algorithm is not ensured to terminate: If an exit node's context depends on itself, we recursively call GetNextExpansion infinitely often, not reaching the base case in Line 8. An example where this happens is in the RSM Figure 2 when checking against the formula $\Phi = \forall \mathsf{X} \exists \bigcirc \mathsf{U} \bullet$ where GetNextExpansion would be called with (b, n_6) and $\phi = \exists \bigcirc \mathsf{U} \bullet$. In the following steps, GetNextExpansion would be invoked with ϕ on n_6, n_7, (b, n_7), n_2, and finally (b, n_6) again. To resolve such cycles, we first keep track of the node-formula pairs for which GetNextExpansion has been already invoked. If a cycle is detected by trying to invoke GetNextExpansion with the same parameters, we backtrack until we can make a different choice in a disjunction- or exists-case, possibly leading to a box to be contextualized. This backtracking procedure is only successful if there is such a box not involved in any dependency cycle. For instance, in the example above such a box does not exist. However, in such a case, similar reasoning as done for global dependency cycle resolution in Section 3.4 can be applied to refine interpretations in nodes of a global dependency cycle.

4.2 Lazy Approach to RSM Model Checking

The idea of lazy contextualization of boxes in an RSM can be incorporated into the exhaustive RSM model-checking approach ExhaustiveCheck pre-

Algorithm 5: LazyCheck($\underline{\mathcal{B}}, \Phi$)

input : RSM $\underline{\mathcal{B}} = (\mathcal{B}_1, \ldots, \mathcal{B}_k)$ and CTL formula Φ, both over AP
output: `tt` if $\underline{\mathcal{A}} \models \Phi$ and `ff` if $\underline{\mathcal{A}} \not\models \Phi$

```
1  A, ∂ := Initialize(B, Φ)
2  F := ∅
3  while F ≠ Subf(Φ) do
4  |  Pick φ ∈ Subf(Φ) with Subf(φ) \ F = {φ}
5  |  F := F ∪ {φ}
6  |  ∂ := GlobalDeduce(A, φ, ∂)
7  if there is en ∈ En_1 with ∂_1(en, Φ) = ff then return ff
8  while there is en ∈ En_1 with ∂_1(en, Φ) = ?? do
9  |  b = GetNextExpansion(A, en, Φ, ∂, ε)
10 |  Contextualize(A, Φ, ∂, b)
11 |  while F ≠ Subf(Φ) do
12 |  |  Pick φ ∈ Subf(Φ) with Subf(φ) \ F = {φ}
13 |  |  ∂ := GlobalDeduce(A, φ, ∂)
14 |  if there is en ∈ En_1 with ∂_1(en, Φ) = ff then return ff
15 return tt
```

sented in Algorithm 3. This leads to a method LazyCheck presented in Algorithm 5. While ExhaustiveCheck surely contextualizes all boxes with contexts encountered during global deduction GlobalDeduce, Algorithm 5 uses GetNextExpansion to contextualize only those boxes that might contribute to deciding whether the target formula Φ holds in the outermost component of the RSM. In essence, Algorithm 5 follows the same reasoning principles as ExhaustiveCheck given in Algorithm 3 by employing functions Initialize, GlobalDeduce, and Contextualize. The main difference is that due to the lazy evaluation of subformulas, the satisfaction of subformulas is not a priori known before invoking a global deduction GlobalDeduce (see Line 13). However, due to our ternary reasoning implemented in LocalDeduce and the progress in contextualizing boxes through GetNextExpansion in combination with the global dependency cycle resolution described in Section 4.1, we obtain correctness and soundness of our new model-checking algorithm for RSMs.

Theorem 2. LazyCheck($\underline{\mathcal{A}}, \Phi$) *terminates for any RSM* $\underline{\mathcal{A}}$ *and CTL formula* Φ *over a common set of atomic propositions and returns* `tt` *iff* $\underline{\mathcal{A}} \models \Phi$.

Proof sketch. The termination and soundness arguments are analogous to the arguments in the proof of Theorem 1 but require an additional step to prove that GetNextExpansion terminates and is sound. This is achieved by careful analysis of the implementation of the cycle resolution described in the last section about global dependency cycle resolution. Termination of the full algorithm LazyCheck then follows from the strict refinements also within adding new contexts, for which there are only finitely many. Soundness follows by the soundness of all subroutines as LazyCheck does not directly modify $\underline{\partial}$. □

Note that in the worst case, all boxes have to be contextualized to determine whether the RSM $\underline{\mathcal{A}}$ satisfies a CTL formula Φ. In this case, our algorithm is also an exhaustive algorithm with a slight polynomial-time overhead of the reasoning steps involved in GetNextExpansion. However, the termination condition might be satisfied after fewer contextualizations as we have seen in our example of Figure 2, resulting in strictly less computation steps than ExhaustiveCheck and illustrating the potential of our lazy model-checking approach.

5 Implementation and Evaluation

We implemented both the exhaustive and the lazy approach presented in this paper in a prototypical tool RSMCheck. Written in Python3, it is supported by almost all common operating systems. RSMs are specified by a dedicated JSON format, to which our tool also provides a translation from pushdown systems for model checkers PDSolver [15] or PuMoC [20] that follows the standard translation method (see, e.g., [6]).

Research questions. To demonstrate applicability of our tool and investigate properties of the algorithms presented in this paper, we conducted several experimental studies driven by the following research questions:

- **(RQ1)** Is our lazy approach effective, i.e., generates significantly less contexts and is faster compared to the exhaustive approach?
- **(RQ2)** How do analysis times of our approaches implemented in RSMCheck compare to state-of-the-art procedural model checkers?
- **(RQ3)** Can real-world procedural programs be verified with our approaches?

Experimental setup. All our experiments were carried out using PyPy 7.3.3 on an Intel i9-10900K machine running Ubuntu 21.04, with a timeout threshold of 30 minutes and a memory limit of 4 GB of RAM.

5.1 Scalability Experiment

First, we conducted a scalability experiment to compare the exhaustive and lazy approach. We randomly generated 2 500 RSM/CTL-formula pairs $(\underline{\mathcal{A}}_i, \Phi_j)$ of increasing sizes and formula lengths: For $i, j \in \{1, \ldots, 50\}$ the RSM $\underline{\mathcal{A}}_i$ contains i components, each having $\lfloor i/3 \rfloor$ boxes and $3i$ nodes with connectivity of 20%, while the formula Φ_j has a quantifier depth of $\lfloor j/9 \rfloor$. Figure 3 shows the analysis times in seconds for our lazy (left) and exhaustive (right) approach. We observe that the more compositional structure and the bigger the requirement formulas, the more the lazy approach pays off compared to the exhaustive approach, both in memory consumption and analysis speed. In 5% of the cases, the exhaustive approach ran into memouts and in all other cases the lazy approach is on average eight times faster than the exhaustive one. For **(RQ1)** we conclude that lazy contextualization is an effective method that allows for faster RSM model checking.

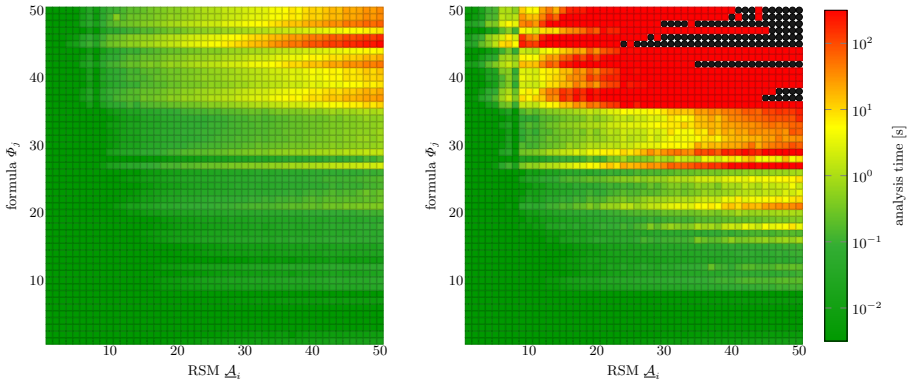

Fig. 3: Analysis times for the scalability experiment in seconds (logarithmic scale, lazy on the left, exhaustive on the right, • marks stand for memouts)

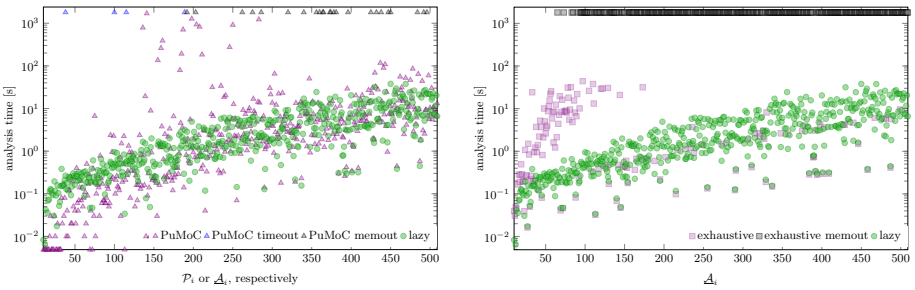

Fig. 4: Analysis times for 500 PuMoC examples in seconds (logarithmic scale)

5.2 PuMoC Benchmark Set

Our second experimental study compares RSMCheck to the procedural CTL model checker PuMoC on its benchmark set [20]. The benchmark set of PuMoC comprises 500 randomly generated pushdown systems $\mathcal{P}_i$ and CTL formulas Φ_i, numbered as in [20] with $i \in \{10, \ldots, 509\}$. Here, the sizes of the pushdown systems increase with increasing i. To enable RSM model checking, we translated each $\mathcal{P}_i$ to an RSM $\underline{\mathcal{A}}_i$ in the input format of RSMCheck. The resulting RSMs have only one component and thus, our lazy approach is expected to not fully use its potential. However, while PuMoC runs into time- or memouts in 28 examples, the lazy approach successfully completes each experiment in less than 40 seconds. Most of the analysis times are in the same range (see Figure 4 on the left) even though PuMoC is implemented in C, while RSMCheck is implemented in Python, known for broad applicability but comparably weak performance. Regarding **(RQ2)**, we can conclude that RSMCheck is competitive with the state-of-the-art model checker PuMoC even on single-component RSMs. Figure 4 on the right shows a comparison of the lazy approach to the exhaustive one, applied on the 500 PuMoC examples. The exhaustive approach

Table 1: Analysis statistics for JAVA interprocedural analysis (time in seconds)

JAVA program	result	PDSOLVER time	PUMOC time	k	exhaustive #ctx	exhaustive time	lazy #ctx	lazy time
Dataflow (Fig. 1)	ff	<0.01	0.02	3	6	<0.01	1	<0.01
avroraCFG	tt	>1 800	>1 800	3 169	4 372	133.41	2	7.46
avroraDisassemble	tt	806.66	>1 800	2 085	4 628	233.18	1	3.19
avroraELF	tt	26.68	71.28	248	614	4.79	1	0.29
avroraMedTest	tt	12.48	37.09	238	264	1.28	4	0.43
avroraReg	tt	8.73	16.12	173	477	1.69	2	0.32
dom2pdf	tt	80.46	1 345.56	615	2 002	17.88	1	0.76
fop2pdf	tt	61.68	>1 800	607	2 029	27.65	6	1.19

is always slower than the lazy approach and runs into memouts in 69% of the cases. This also supports our positive answer to **(RQ1)** drawn in the last section.

5.3 Interprocedural Static Analysis for Java Programs

Our last experimental study considers an interprocedural analysis for real-world systems, borrowed from the benchmark set of [15]. These benchmarks comprise pushdown systems modeling the control-flow of JAVA programs with use-def annotations for all variables of the program, allowing for a data-flow analysis of the program. We first used our implementation to translate programs and the annotated requirement from the input formalism of PDSOLVER to the input formalisms of PUMOC and RSMCHECK. The requirement formalizes that whenever the selected variable is defined, it is eventually used (see the Dataflow example in the preliminaries). Table 1 shows characteristics of our analysis. First, the lazy and even the exhaustive approach are significantly faster than PDSOLVER and PUMOC. Thus, contributing to **(RQ2)** and **(RQ3)**, RSMCHECK can be faster than state-of-the-art procedural model checkers also on real-world models. This can be explained by the compositional structure of RSMs and their generation of contexts: Even the exhaustive approach generates only those contexts that arise during deduction steps in exit nodes. These studies also support that our lazy approach is effective (cf. **(RQ1)**): Column k of Table 1 indicates the number of components of the RSM for the JAVA program, while #ctx indicates the number of generated contexts during analysis. We can observe that the lazy approach effectively avoids context generation, having a direct impact on the analyzed state spaces and timings. Further, we observe speedups of up to two orders of magnitude compared to the exhaustive approach.

6 Conclusion and Discussion

We presented a novel technique to model check RSMs against CTL requirements, combining ternary reasoning with lazy contextualization of components. While of heuristic nature, our experimental studies showed significant speedups compared to existing methods in both scalability benchmarks and in an interprocedural data-flow analysis on real-world systems. Our tool RSMCheck is, to the best of our knowledge, the first tool that implements the RSM model-checking approach by Alur et al. [1] for verifying CTL formulas.

Counterexamples and witnesses. One major advantage of model-checking approaches is the generation of counterexamples or witnesses for refuting or fulfilling the analyzed requirement, respectively. Also in RSMCheck we implemented a witness-generation method that traverses the nodes of the RSM according to computed interpretations similarly as GetNextExpansion does to find a path responsible for requirement satisfaction. The main difference to the standard witness-generation methods in Kripke structures is that not only nodes are tracked but also call stacks and contexts. Counterexamples for universally quantified requirements are obtained by our witness-generation method applied on the complement existential requirement.

Expansion heuristics. Central in our lazy approach is the nondeterministic algorithm GetNextExpansion, which determines the next context to be considered. This algorithm leaves some freedom in how the nondeterminism is resolved, for which plenty of heuristics are reasonable. We implemented two methods, a random selection of subformulas and a deterministic selection that chooses the left-most unknown subformula for further recursive calls, e.g., in the disjunctive case in Line 2 of Algorithm 4. The latter is set as default to enable developers to control the verification process by including domain knowledge, e.g., by placing most influential subformulas upfront to further exploit lazy context evaluation. In our experimental studies, choosing either heuristic did not significantly change runtimes, which is explainable since the CTL requirements were either randomly generated or a comparably simple use-def formula.

Related work. The most commonly used state-based formalisms for procedural programs are pushdown systems (PDSs) and RSMs, for which there are linear-time transformations that lead to bisimilar Kripke structure semantics [2]. While PDSs take a more theoretical perspective, essentially encoding pushdown automata, RSMs directly reflect the programs procedural structure. Model checkers for procedural programs have been first-and-foremost implemented for PDSs, ranging from PuMoC [20] for CTL requirements and PDSolver [15] for requirements specified in the CTL-subsuming μ-calculus, to the LTL model checker Moped [19] also integrated into PuMoC. The latter relies on a symbolic engine that uses *binary decision diagrams (BDDs)* [8], shown to be beneficial for LTL model checking on large-scale procedural programs [19]. On-demand or lazy approaches for interprocedural analysis have been considered, e.g., to determine evaluation points for a priori narrowed scopes [16], or to analyze the interplay be-

tween classes and objects in JavaScript programs [17]. Contrary, our approach focuses on lazy verification on state-based models.

Further work. In next development steps, we plan to also include the support for CTL* requirements, using well-known automata-theoretic constructions for LTL model checking (see, e.g., [1,4]). Further, we plan to extend RSMCheck with a BDD-based model-checking engine to investigate the impact of our lazy algorithms also in the symbolic setting. Remind that our experiments showed that explicit lazy model checking is already efficient on large real-world systems where state-of-the-art (symbolic) procedural model checkers were not able to complete the verification process. Many extensions for PDSs have been presented in the literature, which could also serve as bases for extending our work on lazy RSM model checking. For instance, *weighted RSMs* (see, e.g., [18]) equip RSMs with labels from a semi-ring, similarly as *probabilistic RSMs* equip transitions with probabilities (see, e.g., [6,13]).

References

1. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T., Yannakakis, M.: Analysis of recursive state machines. ACM Trans. Program. Lang. Syst. **27**(4), 786–818 (Jul 2005)
2. Alur, R., Bouajjani, A., Esparza, J.: Model Checking Procedural Programs, pp. 541–572. Springer International Publishing, Cham (2018)
3. Alur, R., Yannakakis, M.: Model checking of hierarchical state machines. ACM Trans. Program. Lang. Syst. **23**(3), 273–303 (2001)
4. Baier, C., Katoen, J.P.: Principles of model checking. The MIT Press (2008)
5. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Proc. of CONCUR'97, LNCS, vol. 1243, pp. 135–150. Springer (1997)
6. Brázdil, T.: Verification of Probabilistic Recursive Sequential Programs. Ph.D. thesis, Masaryk University Brno (2007)
7. Bruns, G., Godefroid, P.: Model checking partial state spaces with 3-valued temporal logics. In: Proc. of CAV'99. pp. 274–287. Springer (1999)
8. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers **35**, 677–691 (1986). https://doi.org/10.1109/TC.1986.1676819
9. Burkart, O., Steffen, B.: Model checking for context-free processes. In: Proc. of CONCUR'92. pp. 123–137 (1992)
10. Burkart, O., Steffen, B.: Model checking the full modal mu-calculus for infinite sequential processes. Theor. Comput. Sci. **221**(1-2), 251–270 (1999)
11. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logic of Programs. LNCS, vol. 131, pp. 52–71 (1981)
12. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (2000)
13. Etessami, K., Yannakakis, M.: Recursive markov chains, stochastic grammars, and monotone systems of nonlinear equations. J. ACM **56**(1) (2009)
14. Fehnker, A., Dubslaff, C.: Inter-procedural analysis of computer programs. US Patent 8,296,735 (2012)

15. Hague, M., Ong, C.H.: A saturation method for the modal μ-calculus over pushdown systems. Information and Computation **209**(5), 799 – 821 (2011)
16. Horwitz, S., Reps, T., Sagiv, M.: Demand interprocedural dataflow analysis. In: Proc. of SIGSOFT'95. pp. 104–115. ACM (1995)
17. Jensen, S.H., Møller, A., Thiemann, P.: Interprocedural analysis with lazy propagation. In: Proc. of Static Analysis. pp. 320–339. Springer (2010)
18. Reps, T.W., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Sci. Comput. Program. **58**(1-2), 206–263 (2005)
19. Schwoon, S.: Model checking pushdown systems. Ph.D. thesis, Technical University Munich, Germany (2002)
20. Song, F., Touili, T.: PuMoC: A CTL model-checker for sequential programs. In: Proc. of ASE'12. pp. 346–349. ACM (2012)