Software and Systems Modeling
https://doi.org/10.1007/s10270-024-01159-z

SPECIAL SECTION PAPER

Lazy model checking for recursive state machines

Clemens Dubslaff[1,2] · Patrick Wienhöft[2,3] · Ansgar Fehnker[4]

Received: 30 June 2022 / Revised: 31 January 2024 / Accepted: 8 February 2024
© The Author(s) 2024

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 have been intensively studied in the literature. In this article, 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 implemented our model-checking algorithms and evaluate them 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.

Keywords Model checking · Lazy verification · Interprocedural static analysis · Recursive state machines · Computation tree logic

1 Introduction

Model checking [6, 20] is a well-established automated verification technique that shows for a system model whether a formal requirement is met or not. 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,* [18]). CTL is an expressive branching-time logic that is well-accepted in the community and covers many of the most important requirement patterns [26]. While model checking is first and foremost applied toward a sound and exhaustive analysis of systems, it has been also shown to be effective for a *static code analysis* by model checking data-flow or control-flow graphs [46]. Static analysis is in particular challenging for *interprocedural properties*, i.e., requirements that exceed the scope of single procedures and objects that consequently depend on a multitude of environments in which the procedures can be called. Several approaches have been proposed to tackle the challenges of an interprocedural analysis, e.g., by introducing property summaries of procedures [10, 48] or by exploiting analysis techniques on operational procedural models by means of *pushdown systems (PDSs,* [45]) or *recursive state machines (RSMs,* [3]).

The scope of this article is in the context of model checking RSMs against CTL properties, e.g., for interprocedural static code analysis. RSMs closely follow the compositional structure of programs with recursive procedure calls by modeling each procedure by a separate Kripke structure (called a *component*). Each component has *entry* and *exit* nodes used to model the input and output functionalities of the corresponding procedure. Procedure calls in RSMs are then formalized through *boxes* that are placeholders for components with the matching *call* and *return* nodes in the calling Kripke structure. To this end, RSMs can be also seen as "nested" Kripke structures. The operational semantics of RSMs arises

Communicated by Antonio Cerone and Frank de Boer.

✉ Clemens Dubslaff
c.dubslaff@tue.nl

✉ Patrick Wienhöft
patrick.wienhoeft@tu-dresden.de

Ansgar Fehnker
ansgar.fehnker@mq.edu.au

1 Eindhoven University of Technology, Eindhoven, The Netherlands

2 Centre for Tactile Internet with Human-in-the-loop, Dresden, Germany

3 Technische Universität Dresden, Dresden, Germany

4 Macquarie University, Sydney, Australia

Published online: 20 March 2024

by replacing boxes with associated components matching entry and exit with call and return nodes, respectively. As components can be called recursively in RSMs (different from *hierarchical state machines* [5]), the semantics of RSMs might have infinitely many states, which renders the standard CTL model-checking algorithm for finite Kripke structures [6, 18] not directly applicable [3]. However, as first noticed in the area of PDSs [15], 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* [5, 15]. Intuitively, contexts model the environmental influence on the component, i.e., how the satisfaction of the formula depends on the calling component. A CTL model-checking algorithm for RSMs then could be almost directly derived by eagerly generating all contexts that could arise during program execution and then applying the standard CTL model-checking algorithm for finite Kripke structures on components [3]. Such an algorithm runs in time exponential in the size of the RSM due to possibly exponentially many contexts that have to be considered for each component. Further, the above direct algorithms employ a decomposition method on executions that does not take the compositional structure of RSMs into account [3], possibly leading to exponentially many break points for executions. Since the model-checking problem for RSMs and CTL formulas is ExpTime-complete [9], this algorithm cannot be improved in the worst case. Nevertheless, there is plenty of room for optimizations. First, one could investigate methods that are fully compositional by means of investigating components depending on their contexts in isolation, also on the execution path level. Second, one might reduce the number of subformulas and contexts to be evaluated by heuristics. Both could provide a more clean approach and also show runtime improvements in practical relevant applications—an opportunity that has, to the best of our knowledge, not yet been considered in the literature.

1.1 Contribution

We contribute to the quest of devising practical algorithms and heuristics that speed up CTL model checking for RSMs. In particular, we make the following contributions:

1. Eager and lazy CTL model-checking algorithms for RSMs that use ternary model checking and path decomposition at the level of components.
2. An implementation of the eager and lazy approaches along with various context expansion heuristics.
3. Evaluation of our algorithms regarding scalability, on community benchmarks, and on real-world Java benchmarks that show superior performance and practical applicability of our algorithms and implementation.

Ad 1: To establish our algorithms, we extend the RSM model-checking approach by Alur et al. [3] to arbitrary RSMs (not bounding the number of exit nodes). Further, we take inspiration from ternary model checking by Bruns and Godefroid [12] toward a ternary component-wise decomposition scheme of executions. We call this generalized method *eager approach*, since with this method, all subformulas and all components with reachable contexts are checked. The idea behind our second method, which we call *lazy approach*, is to adjoin a lazy evaluation scheme that successively refines the ternary global satisfaction relation by step-wise evaluating contexts and subformulas that could contribute to deciding the overall model-checking problem [28]. To this end, the lazy approach can avoid to check subformulas and certain components with their contexts if the satisfaction of a formula is already determined independent of such.

Ad 2: We implemented both, the eager and lazy approach, in a tool called RSMCheck[1]. 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 [33] and PuMoC [49] rely on PDSs. Besides RSMs, RSMCheck can also model check PDS input formats of PDSolver and PuMoC via conversion scripts we implemented to transform PDSs to RSMs based on the well-known linear-time transformation that preserves the Kripke structure semantics [11]. 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 facilitates the lazy approach can ease the interpretation of counterexamples and witnesses generated by model checking and hence also supports debugging during program development steps.

Ad 3: We conduct three experimental studies for RSMCheck, addressing scalability, comparison to existing model-checking tools, and application to real-world examples by means of an interprocedural data-flow analysis on Java programs. In these studies we show that our eager and lazy approaches are effective, where the lazy one evaluates less contexts than in the eager case, leading to significant speedups up to one order of magnitude. Applied on their own benchmark suites, PDSolver and PuMoC show timeouts or exceed memory constraints on several instances [49]. We demonstrate that our lazy approach manages to verify all instances and outperforms PDSolver and PuMoC by being up to two orders of magnitude faster.

[1] The tool can be downloaded at https://github.com/PattuX/RSMCheck. A static reproduction package detailing how to reproduce our experimental studies can be downloaded at https://osf.io/h2f5u/

1.1.1 Disclaimer

This article is based on the conference publication titled "Be Lazy and Don't Care: Faster CTL Model Checking for Recursive State Machines" [25]. Besides full proofs of the correctness of the algorithms presented, an additional modular formalization and evaluation of contextualization heuristics, discussions on problem instances with their impact on algorithmic complexities, an improved implementation of the eager approach, we provide further explanations, examples, and descriptions that underpin the overall approach.

1.1.2 Outline

After settling notations and basic definitions in Sect. 3, we first extend the model-checking approach by Alur et al. [3] to the multi-exit and ternary setting in Sect. 5. Our lazy approach is detailed in Sect. 6 and evaluated in Sect. 7 for different heuristics. We close the paper with concluding remarks and future work in Sect. 8. In the appendix, we provide the full proofs of theorems and lemmas.

2 Related work

In this section, we relate our work to the literature. For general overviews about temporal logics and model checking we refer to standard textbooks [6, 21].

2.1 Recursive operational models

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 [4]. 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 [49] for CTL requirements and PDSolver [33] for requirements specified in the CTL-subsuming μ-calculus [39], to the LTL model checker Moped [47] also integrated into PuMoC. The latter relies on a symbolic engine that uses *binary decision diagrams (BDDs,* [14]), shown to be beneficial for LTL model checking on large-scale procedural programs [47].

There are several other related hierarchical modeling formalisms for which model-checking methods have been established, e.g., *hierarchical state machines* [5], *nested Kripke structures* [29], and *incomplete Büchi automata* [44]. They all do not support infinite recursion and hence do not face the challenges we address in our approach. However, those approaches could surely benefit also of our lazy evaluation schemes presented in this article.

2.1.1 Abstraction and refinement

Abstraction methods and stepwise refinement constitute key techniques to provide scalable verification [24], ranging from, e.g., *abstract interpretation* [22, 23], predicate abstraction [32], and *counterexample-guided abstraction refinement* (CEGAR, [19]) to multi-valued modeling and verification on abstracted models [17].

Most related to our work is the latter view on abstraction, i.e., explicitly modeling parts of systems as "unknown" or "uncertain" and perform ternary logic reasoning. *Modal transition systems (MTSs)* [41] model underspecification or incomplete information at the level of transitions, distinguishing transitions that must be, possibly may, or are surely not present in an actual implementation (cf., e.g., a survey on MTSs [40]). Verification of MTSs mainly has been investigated for formulas expressed in the modal μ-calculus [30] with action labels, but also state-labeled extensions by means of *Kripke MTSs* [36, 37]. Ternary CTL model-checking of Kripke structures, as we also do in in this paper, has been first considered to reason about partial state spaces, formalized as *partial Kripke structures (PKSs)* [12]. Besides the ternary semantics of formulas and PKSs [12], the alternative *thorough semantics* allows for deducing more definitive labels where the classical semantics would not come to a conclusion [13]. While interesting by its own, we chose the original semantics [12]: thorough evaluation is computational more expensive—ExpTime-complete [13]—while classical ternary model checking is doable in polynomial time. Further, the corner cases can usually be treated by formula simplifications (e.g., resolving tautologies) when transforming CTL formulas into existential normal form.

Within PKSs, full information on the system topology is assumed, only supporting partial information on the state labelings. The recursive state machines we consider in this article have a different nature, allowing for a component structure where whole parts of the system are not evaluated. However, PKSs and ternary evaluations already show great applicability also in software refinement cycles, e.g., in combination with proof assistance [7, 43]. Verification on partial behavioral models such as PKSs can be used as basis for incremental model-based software engineering and analysis [50].

2.1.2 Lazy evaluation approaches

Opposed to classical *eager* abstraction and refinement, where reasoning steps are exhaustively performed, on-demand or *lazy approaches* postpone abstraction and refinement until it is inevitable toward drawing conclusions. A generic

abstract-check-refine approach with lazy evaluation has been presented through *lazy abstraction* [34], which, however, did not support infinite state spaces as we have to deal with in RSMs. This work has been extended to support infinite state spaces by using interpolants instead of predicate abstraction [32], also fastening the lazy evaluation process [42, 51]. However, their work approaches model checking more from a SAT-perspective and does not take the potential compositional structure of RSMs into account, as we do.

Lazy approaches for interprocedural analysis have been considered, e.g., to determine evaluation points for a priori narrowed scopes [35], or to analyze the interplay between classes and objects in JAVASCRIPT programs [38]. Contrary, our approach focuses on lazy verification on state-based models.

3 Preliminaries

We first settle notations and build the formal framework for recursive state machines and computation tree logic we use throughout this paper.

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. An *interpretation* over X is a function $\partial\colon X \to \{\mathbf{tt}, \mathbf{ff}, \mathbf{??}\}$ where **tt** stands for "true", **ff** for "false", and **??** 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 $\partial'(x) = \mathbf{tt}$ implies $\partial(x) = \mathbf{tt}$, and $\partial'(x) = \mathbf{ff}$ implies $\partial(x) = \mathbf{ff}$. Further, ∂ is a *strict* refinement of ∂' if additionally $\partial \neq \partial'$.

A *Kripke structure* (see, e.g., [6]) 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 $\Pi(s)$.

3.1 Recursive state machines

A labeled *recursive state machine* (RSM, [3]) over a set of atomic propositions AP is a tuple $\underline{\mathcal{A}} = (\mathcal{A}_1, \ldots, \mathcal{A}_k)$ comprising *components*

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

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, defining the following auxiliary sets:
  - the set of *call nodes* of a box $Call_b \stackrel{\text{def}}{=} \{(b, en) \mid b \in B_i, \text{ and } en \in En_{Y_i(b)}\}$
  - the set of *call nodes* of a component $Call_i \stackrel{\text{def}}{=} \bigcup_{b \in B_i} Call_b$
  - the set of *return nodes* of a box $Return_b \stackrel{\text{def}}{=} \{(b, ex) \mid b \in B_i, \text{ and } ex \in Ex_{Y_i(b)}\}$
  - the set of *return nodes* of a component $Return_i \stackrel{\text{def}}{=} \bigcup_{b \in B_i} Return_b$
  - the *total node set* $\mathsf{N}_i^{all} \stackrel{\text{def}}{=} N_i \cup Call_i \cup Return_i$
- $\longrightarrow_i \subseteq (N_i \setminus Ex_i) \cup Return_i \times (N_i \setminus En_i) \cup Call_i$ is a transition relation, and
- AP is a set of atomic propositions,
- $L_i\colon \mathsf{N}_i^{all} \to \wp(AP)$ is a node labeling function for which $L_i((b, n)) = L_{Y_i(b)}(n)$ for all $(b, n) \in Call_i \cup Return_i$.

For brevity, we will omit indices from the parts of a component to refer to the union over all component, e.g., $N \stackrel{\text{def}}{=} \bigcup_{i=1}^k N_i$ or $B \stackrel{\text{def}}{=} \bigcup_{i=1}^k B_i$. Lastly, we extend the transition relation $trans$ to toward containing transitions into components as follows:

$$\longrightarrow_+ \stackrel{\text{def}}{=} \longrightarrow \cup \bigcup_{i=1}^{k} \big\{((b, n), n) \mid b \in B_i, n \in En_{Y(b)}\big\}$$

We assume that all nodes except exit nodes are not final, i.e., for all $i \in \{1, \ldots, k\}$ and $n \in (N_i \setminus Ex_i) \cup Return_i$ there is $n' \in (N_i \setminus En_i) \cup Call_i$ such that $n \longrightarrow_i n'$. Note that we allow for direct transitions from return to call nodes.
The semantics of a component $\mathcal{A}_i$ is defined as Kripke structure $[\![\mathcal{A}_i]\!] = (\mathsf{N}_i^{all}, \longrightarrow_i, AP, L_i)$. The semantics of $\underline{\mathcal{A}}$ is a Kripke structure

$$[\![\underline{\mathcal{A}}]\!] \;=\; (B^* \times \mathsf{N}^{all}, \Longrightarrow, AP, L)$$

where L labels each state as the corresponding node, i.e., $L((\sigma, n)) = L_i(n)$ for all $\sigma \in B^*$ and $n \in \mathsf{N}_i^{all}$, and $\Longrightarrow$ is the smallest transition relation that obeys the rules of Fig. 2. 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.

```
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 }
```

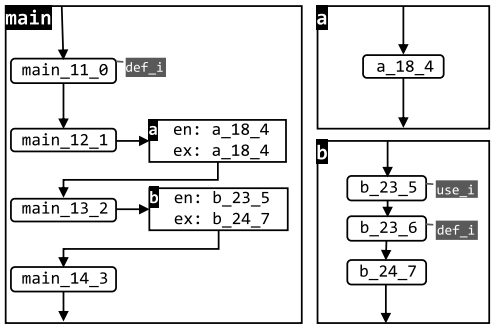


Fig. 1 JAVA dataflow example [33] and its generated control-flow RSM with components `main`, `a` and `b`. Rounded rectangles represent nodes with labels attached next to them on gray background. Boxes are represented by rectangles with the box label giving their referenced component in the top left. Transitions are denoted by arrows

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

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

Fig. 2 Kripke structure semantics of recursive state machines

Example 1 Fig. 1 depicts a JAVA program (left) and a corresponding 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 $\mathtt{use}_i$ and $\mathtt{def}_i$, indicating whether variable i is read or written, respectively.

3.2 Computation tree logic

We specify system requirements formulas of *computation tree logic* (CTL, [18]), which are defined over atomic propositions AP by the grammar

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

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 [6]. We denote by $Subf(\Phi)$ and $Subf_\exists(\Phi)$ the set of subformulas and existential subformulas of Φ, respectively. We say a formula $\phi \in Subf(\Phi)$ is a *strict* subformula of Φ if $\phi \neq \Phi$. Given a Kripke structure $\mathcal{K} = (S, \longrightarrow, AP, L)$, we define the satisfaction relation $\models$ for CTL formulas over AP recursively by

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

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

Example 2 The dataflow example RSM provided in Example 1 could be subject of an interprocedural data-flow analysis employing its use-def annotations. For instance, the requirement that whenever the variable i is defined, it is eventually used, can be expressed by the CTL formula

$$\Phi \ = \ \forall \mathsf{G}\big(\mathtt{def}_i \rightarrow \exists \mathsf{F}(\mathtt{use}_i)\big).$$

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 Fig. 1, this is witnessed by the only existing execution that starts in the initial node main_11_0, reaches the $\mathtt{def}_i$-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 $\mathtt{use}_i$.

4 Problem statement

In this section, we formalize the model-checking problem for RSMs and CTL properties that we target in this paper, and elaborate more on the main challenges that have to be tackled.

For a CTL formula Φ, we write $\underline{\mathcal{A}} \models \Phi$ if for all entry nodes of the outermost component $n \in En_1$ can ensure satisfaction of Φ, i.e., $(\varepsilon, n) \models \Phi$ in the Kripke structure semantics $[\![\underline{\mathcal{A}}]\!]$ of the $\underline{\mathcal{A}}$ [15, 16]. The *model-checking prob-*

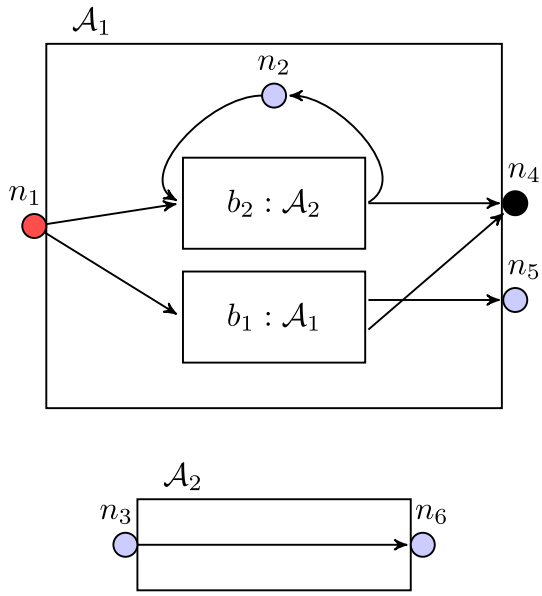


Fig. 3 Running example RSM

lem we consider here in this paper then asks whether $\underline{\mathcal{A}} \models \Phi$ for a given RSM $\underline{\mathcal{A}}$ and CTL formula Φ, both over AP.

Example 3 Fig. 3 depicts an RSM $\underline{\mathcal{A}} = (\mathcal{A}_1, \mathcal{A}_2)$ over $AP = \{\bigcirc, \bullet, \bullet\}$. Initially, there is a choice between again entering $\mathcal{A}_1$ or entering $\mathcal{A}_2$. When entering $\mathcal{A}_1$ via b_1, this choice can be repeated arbitrarily many times, leading to a box stack filled with arbitrarily many b_1's. Note that this behavior includes the infinite case of unboundedly many times re-entering $\mathcal{A}_1$. Otherwise, when entering $\mathcal{A}_2$ via b_2, we have to leave $\mathcal{A}_2$ directly after one step. After leaving, there is a choice to re-entering $\mathcal{A}_2$ after visiting n_2, which can also be done arbitrarily many times, leading to an increasing box stack but now with boxes b_2. Not entering $\mathcal{A}_2$ again leads to transitioning to n_4, leaving $\mathcal{A}_1$ through this exit node. This reduces the box stack by b_1, transitioning from return node (b_1, n_5) to exit node n_4 or from return node (b_1, n_4) to exit node n_5, alternating whether we leave b_1 via n_4 or n_5 until the box stack is empty. Due to this alternation, the final node depends on the parity of the number of times b_1 has been entered.

Consider the CTL property $\exists\mathsf{G}\bullet$, which is satisfied for $\underline{\mathcal{A}}$, witnessed by the infinite run $(\epsilon, n_1)(\epsilon, (b_1, n_1))(b_1, n_1)(b_1, (b_1, n_1))(b_1 b_1, n_1)\ldots$. To prove satisfaction, a naive attempt would be to flatten the $\underline{\mathcal{A}}$ into a Kripke structure, i.e., recursively replacing all boxes by their references component. However, this flattened Kripke structure is infinite, since $b_1 \in B_1$ calls $\mathcal{A}_1$, leading to an infinite recursion. To circumvent exhaustive flattening, we could flatten $\underline{\mathcal{A}}$ up to a certain box stack size, obtaining a finite *partial Kripke structure (PKS)* [7, 13, 43, 44]. Nevertheless, since the property can only be fulfilled on models with infinite runs, finite PKSs cannot directly verify satisfaction of the property.

One possible approach to solve the model-checking problem on the level of PKSs would be to find a depth bound k such that after flattening boxes up to the depth k we can use interpolation techniques toward complete verification [42, 51]. However, this process of flattening loses information about the compositional structure. To provide a compositional approach, we will propose methods that operate directly on the component structure. This allows us to do reasoning locally on the components and also eases explanation of witnesses and counterexamples when referring back to the actual code the RSM was constructed from.

4.1 Contexts

As illustrated in Example 3, one of the main challenges within RSM model checking is the potentially infinite state space of the underlying Kripke structure. Fortunately, previous work on verifying CTL formulas Φ in components showed that it is sufficient to consider finitely many *contexts* to determine the complete satisfaction relation [5, 15]. Contexts comprise all possible (but finitely many) satisfactions of subformulas in return nodes of components. Formally, for an RSM $\underline{\mathcal{A}}$ and a CTL formula Φ both over a set of atomic propositions AP as formalized in Sect. 3, a Φ-*context* of a component $\mathcal{A}_i$ in $\underline{\mathcal{A}}$ is 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. For $\underline{\mathcal{A}} = (\mathcal{A}_1, \ldots, \mathcal{A}_k)$ we again denote the union over all contexts by $\underline{\gamma} \stackrel{\text{def}}{=} \bigcup_{i=1}^{k} \gamma_i$.

Intuitively, an existential formula $\Phi = \exists\psi$ and an exit node $ex \in Ex_i$ a context γ_i specifies whether there is a path that satisfies the path formula ψ when leaving the component $\mathcal{A}_i$ through ex. Note that we do not need to consider other subformulas than such existentially quantified ones, since we assume Φ to be in existential normal form (cf. Section 3.2) and other subformulas' satisfaction can be locally determined or directly depend on existential formulas. Hence, we identify contexts with sets of existentially quantified subformulas.

The challenge on the computational side is that in general the number of possible contexts is exponential, both in the length of the CTL formula as well as in the number of exit nodes. While the ExpTime-completeness of the model-checking problem on RSMs and CTL formulas [3] reduces hope toward an efficient algorithm, there are many examples where the number of relevant contexts is much smaller. This motivates the main goal of our approaches to be developed: computing as few contexts as possible to speed up verification.

Example 4 Consider again the RSM in Example 3, and the CTL property $\Phi = \exists\mathsf{F}\exists\mathsf{G}\bullet$, which holds iff we reach n_4 with the empty box stack. Then, there are two existential subformulas: Φ itself and $\phi_1 = \exists\mathsf{G}\bullet$. Since each pair or existential subformula and exit node must be mapped to either **tt** or **ff**,

this yields $2^{|Ex_i|*|Subf_\exists(\Phi)|}$ different contexts for component $\mathcal{A}_i$, i.e., 16 possible contexts for $\mathcal{A}_1$, and 4 possible contexts for $\mathcal{A}_2$.

Notice however, that in $\mathcal{A}_1$, regardless of the box stack, ϕ_1 cannot hold in n_5. Only in n_4 the formula ϕ_1 can be satisfied in the case where the box stack is empty, a case in which also Φ immediately holds in n_4. Further, due to the structure of the RSM, if Φ holds in n_5 for some box stack, then it cannot hold in n_6 for the same box stack and vice versa. In total, this leaves us with only 3 possible fully specified contexts in $\mathcal{A}_1$: one where only (n_4, Φ) is mapped to **tt**, one where only (n_5, Φ) is **tt**, and one where (n_4, Φ_1) and (n_4, ϕ_1) are both mapped to **tt**. Similarly, $\mathcal{A}_2$ has only 2 consistent contexts since ϕ_1 cannot hold in n_6, leading to a total number of 5 consistent contexts out of 20 possible ones.

5 Ternary RSM model checking

This section provides the foundations for a ternary model-checking algorithm of multi-exit RSMs against CTL formulas. Our approach is an adaptation of the CTL* model-checking algorithm for single-exit RSMs by Alur et al. [3] toward ternary reasoning and support multi-exit RSMs. Multi-exit RSMs are RSMs where components might have more than one exit node, which constitutes a large class of real-world procedural programs, not being able to model with single-exit RSMs. For instance, different return values of procedures naturally appear in practical programming examples: except the Dataflow example from Fig. 1, all examples we consider in our experimental studies of Sect. 7 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 and other standard practical patterns [26].

5.1 Motivation

Our reasoning for employing ternary model-checking is that we want to exploit the compositional structure of the RSM. While the algorithm by Alur et al. [3] also computes only reachable contexts, it does so by computing all decompositions of path formulas. The drawback is that the number of decomposition of a formula may be exponential in its length. Further, the computation of decompositions does not exploit the structure of the RSM, neither locally nor globally. Hence, we do not use path formula decompositions but refinement of ternary contexts as an alternative approach. Other reasons for the ternary model-checking procedure we present is that it can provide a basis for lazy evaluation schemes and ease cycle detection, two challenges we face in the forthcoming sections.

Ternary interpretations arise naturally in RSMs as a node satisfying a property may depend on the path after leaving the node's component, which may be unknown until a context is provided. The intuitive semantic meaning of a ternary interpretation that assigns **??** to CTL formula ϕ in node n could be phrased as "The local information about the component and the information provided through contexts is not sufficient to determine whether ϕ holds in n". In particular, this does not necessarily mean that it depends on the box stack whether n satisfies ϕ. It could also mean that $(n, \sigma) \models \phi$ for all (or no) box stacks $\sigma \in B^*$, but the relevant information for determining this has not yet been added to the context of n's component.

5.1.1 Consistency

To formalize the correctness of our algorithms, we introduce the following notions: Given an RSM $\underline{\mathcal{A}}$, a CTL formula Φ, and a ternary interpretation $\underline{\partial}$ over all node-formula pairs $\mathsf{N}^{all} \times Subf(\Phi)$, we say that $\underline{\partial}$ is *consistent* (w.r.t. $\underline{\mathcal{A}}$) iff the following conditions hold for all $n \in \mathsf{N}^{all}$, $\phi \in Subf(\Phi)$, and $\sigma \in B^*$:

$$\underline{\partial}(n, \phi) = \mathbf{tt} \implies (\sigma, n) \models \phi$$
$$\underline{\partial}(n, \phi) = \mathbf{ff} \implies (\sigma, n) \not\models \phi$$

If both implications also hold the other way around, we say that $\underline{\partial}$ is a *maximally consistent*. Note that since both statements quantify universally over stacks $\sigma \in B^*$, it is still possible that a maximally consistent ternary interpretation maps some inputs to **??**. That is, a ternary interpretation with $\underline{\partial}(n, \phi) = \mathbf{??}$ may still be maximally consistent if there are box stacks $\sigma_1, \sigma_2 \in B^*$ with $(\sigma_1, n) \models \phi$ but $(\sigma_2, n) \not\models \phi$.

The support of ternary CTL model checking follows the ideas by Bruns and Godefroid [12] and replaces the role of refinement operations on satisfaction sets in the classical CTL model-checking algorithm for RSMs [3]. To ensure compositional RSM model checking, we discuss two kinds of deductions: first, how ternary interpretations are refined locally for an RSM, and second, how the ternary refinements are globally propagated through contexts.

5.2 Ternary refinement

Toward a complete ternary CTL model-checking algorithm, we first discuss how we refine ternary interpretations for a given RSM. As described earlier, the value of formula in a node may depend on global information, such as the context in which the node's component is called. Hence, it is possible that locally some values are unknown. For this reason, we introduce a ternary model-checking approach that operates on locally known transitions, i.e., transitions within a component and transitions that enter a box. For transitions leaving

$$
\begin{aligned}
[n_0 \models a]_{\underline{\gamma}} &= \begin{cases} \mathbf{tt} & \text{if } a \in L(n_0) \\ \mathbf{ff} & \text{otherwise} \end{cases} \\
[n_0 \models \neg\phi]_{\underline{\gamma}} &= comp([n_0 \models \phi]_{\underline{\gamma}}) \\
[n_0 \models \phi_1 \vee \phi_2]_{\underline{\gamma}} &= \max([n_0 \models \phi_1]_{\underline{\gamma}}, [n_0 \models \phi_2]_{\underline{\gamma}}) \\
[n_0 \models \exists \mathsf{X}\phi]_{\underline{\gamma}} &= \begin{cases} \underline{\gamma}(n_0, \exists \mathsf{X}\phi) & \text{if } n_0 \in Ex \\ \max\{[n \models \phi]_{\underline{\gamma}} \mid n_0 \longrightarrow_+ n\} & \text{otherwise} \end{cases} \\
[n_0 \models \exists \mathsf{G}\phi]_{\underline{\gamma}} &= \max\Big(\Big\{\min\Big\{[n_j \models \phi]_{\underline{\gamma}} \mid 0 \le j \le l\Big\} \mid n_0 \longrightarrow_+ n_1 \longrightarrow_+ \cdots \longrightarrow_+ n_l \longrightarrow_+ n_0\Big\} \cup \\
&\qquad \Big\{\min\Big(\Big\{[n_j \models \phi]_{\underline{\gamma}} \mid 0 \le j < l\Big\} \cup \{\underline{\gamma}(n_l, \exists \mathsf{G}\phi)\}\Big) \mid n_0 \longrightarrow_+ n_1 \longrightarrow_+ \cdots \longrightarrow_+ n_l \text{ and } n_l \in Ex\Big\}\Big) \\
[n_0 \models \exists \phi_1 \mathsf{U} \phi_2]_{\underline{\gamma}} &= \max\Big(\Big\{\min\Big(\Big\{[n_j \models \phi_1]_{\underline{\gamma}} \mid 0 \le j < l\Big\} \cup \{[n_l \models \phi_2]_{\underline{\gamma}}\}\Big) \mid n_0 \longrightarrow_+ n_1 \longrightarrow_+ \cdots \longrightarrow_+ n_l\Big\} \cup \\
&\qquad \Big\{\min\Big(\Big\{[n_j \models \phi_1]_{\underline{\gamma}} \mid 0 \le j < l\Big\} \cup \{\underline{\gamma}(n_l, \exists \phi_1 \mathsf{U} \phi_2)\}\Big) \mid n_0 \longrightarrow_+ n_1 \longrightarrow_+ \cdots \longrightarrow_+ n_l \text{ and } n_l \in Ex\Big\}\Big)
\end{aligned}
$$

Fig. 4 Ternary semantics of CTL formulas in RSMs

a component, we rely on information provided through its context.

Formally, we inductively define the value of a ternary formula Φ in a node $n_0 \in \mathsf{N}^{all}$ by adapting the *ternary semantics* $[n_0 \models \Phi]$ by Bruns and Godefroid [12]. The rules for the resulting ternary semantics $[n_0 \models \Phi]_{\underline{\gamma}}$ including the information provided through context $\underline{\gamma}$ are depicted in Fig. 4.

Here, *comp* is the complement function which maps **tt** to **ff**, **ff** to **tt**, and **??** to **??**. Both min and max are defined on the order **tt** > **??** > **ff**. Our definition extends the usual definition of ternary semantics for cases where Φ is an existentially quantified formula by not only quantifying over all paths in the (partial) Kripke structure that satisfy Φ in the usual way, but also quantifying over all paths that lead to an exit node and considering the context in the exit node. Specifically, CTL formulas of type $\exists\mathsf{X}$ can easily be defined as usual for non-exit nodes since all successors are known, but we exclusively have to rely on the context $\underline{\gamma}$ to determine whether the formula holds in an exit node since we do not know its successors without contextual information. For formulas of type $\Phi = \exists\mathsf{G}\phi_1$ we do not only check for the standard conditions that directly serve as witness (i.e., a cycle on which ϕ_1 holds) but also whether an exit node n_l in which the value of Φ is specified through the context is reachable via a ϕ_1-path. Similarly, for $\Phi = \exists\mathsf{G}\phi_1$ we additionally check we also check whether such an exit node is reachable via a ϕ_1-path. In the ternary setting, for all paths we consider the minimum value over the ternary semantics. As an example, for $\Phi = \exists\mathsf{G}\phi_1$ this means that as soon as a node on the path does not satisfy ϕ_1, or the path reaches an exit node in which Φ does not hold, the path can no longer serve as a witness for Φ. As soon as ϕ_1 is **??** or **tt** in all nodes on the path, and for paths which end in an exit node also the context specifies that Φ is **??** or **tt** in the exit node, the path may serve as a witness for Φ maybe holding. Only if ϕ_1 surely holds on the entire path (and potentially in the exit node through the context), the path serves as a witness for Φ surely holding. Since the formula is existentially quantified we then take the maximum value of a path over all such paths.

In contrast to the common application of this ternary semantics in PKSs, where uncertainty is modeled by $[n \models \Phi] =$ **??** based on a labeling function L, in RSMs the uncertainty comes from a context $\underline{\gamma}$ evaluating an existential formula to **??** in an exit node.

Lemma 1 *Let $\underline{A}$ be an RSM with a consistent context $\underline{\gamma}$ over a CTL formula Φ. Then the ternary interpretation induced by the ternary semantics given by*

$$\underline{\partial}(n, \phi) \;=\; [n \models \phi]_{\underline{\gamma}}$$

for all $n \in \mathsf{N}^{all}$ and $\phi \in Subf(\Phi)$ is consistent.

5.2.1 Maximal consistency

While Lemma 1 shows that the interpretations induced by ternary semantics are consistent, we cannot achieve *maximal consistency*: for a node n and CTL formula Φ, the ternary semantics may evaluate $[n \models \phi] =$ **??**, even if $\Phi \models n$ can be determined locally. This is due to the ternary semantics defined separately considering subformulas and not taking a global formula perspective into account, checking whether there is an interpretation that fulfills or refutes the whole formula.

Example 5 Consider node $n_5 \in N_1$ in the RSM in Example 3, and the CTL property $\Phi = \exists\mathsf{F}\exists\mathsf{G}\bullet \vee \neg\exists\mathsf{F}\exists\mathsf{G}\bullet$ and assume

the context $\underline{\gamma}$ maps all node-formula pairs to **??**. As it cannot be determined locally by only $\mathcal{A}_1$ along with its context γ_1 whether $\exists\mathsf{F}\exists\mathsf{G}\bullet$ holds in n_5, we have $[n_5 \models \exists\mathsf{F}\exists\mathsf{G}\bullet]_{\underline{\gamma}} = \mathbf{??}$. By definition we also have $[n_5 \models \neg\exists\mathsf{F}\exists\mathsf{G}\bullet]_{\underline{\gamma}} = comp(\mathbf{??}) = \mathbf{??}$. Finally, we have $[n_5 \models \Phi]_{\underline{\gamma}} = \max(\mathbf{??}, \mathbf{??}) = \mathbf{??}$. However, Φ is clearly a tautology and thus trivially holds in n_5.

This phenomenon arises in many different applications of ternary model checking and has led to the introduction of the *thorough semantics* [7, 13, 43]. Intuitively, thorough semantics is not defined inductively, but rather checks whether there is an extension of a PKS in which the formula is satisfied, and whether there is another extension in which the formula is not satisfied. It is possible to also adapt the thorough semantics toward RSMs by incorporating information through contexts. In particular, one could check whether there exist consistent refinements of the context such that the formula is satisfied or violated, respectively.

However, while thorough semantics would directly ensure maximal consistency also in RSMs, we decided for the standard semantics. This has mainly two reasons: First, the computation of the value of a formula in a node is computational expensive, i.e., ExpTime-complete even for PKSs [13]. This is further amplified by the fact that the size of the PKS induced by the RSM may be exponential in the size of the RSM (cf. Section 4). Second, maximal consistency can also be ensured when fixing contexts, a property which is crucial for providing a correct model-checking algorithm for RSMs and CTL:

Lemma 2 *Let $\underline{\mathcal{A}}$ be an RSM with context $\underline{\gamma}$ over a CTL formula Φ such that $\underline{\gamma}(n, \phi) \neq$ **??** for all $n \in Ex$ and $\phi \in Subf_{\exists}(\Phi)$. Then the ternary interpretation induced by the ternary semantics given by*

$$\underline{\partial}(n, \phi) = [n \models \phi]_{\underline{\gamma}}$$

for all $n \in \mathsf{N}^{all}$ and $\phi \in Subf(\Phi)$ is maximally consistent.

We will later show that we can always refine contexts in such a way that they never map to **??** which allows us to apply this Lemma.

There is a crucial difference between our application of ternary model checking and the application of PKS mainly considered in the literature/ For RSMs we can always refine contexts, which are the only source of uncertainty in our model, and ultimately are able to remove all uncertainty, while in PKSs uncertainty is part of the model specification. Herein lies the reason why the ternary semantics is sufficient for our approach while the thorough semantics is required in other ternary model checking algorithms [7, 43].

5.2.2 Computation

To compute the value of a given ternary formula Φ on an RSM under the ternary semantics, we use the function RefineTernary($\underline{\mathcal{A}}, \Phi, \underline{\partial}, \underline{\gamma}$). It maps an RSM $\underline{\mathcal{A}} = (\mathcal{A}_1, \ldots, \mathcal{A}_k)$ as in Sect. 3.1, a CTL formula Φ, both over AP, the context $\underline{\gamma}$ of $\underline{\mathcal{A}}$, and an interpretation $\underline{\partial}$ that is consistent with $\underline{\mathcal{A}}$ and $\underline{\gamma}$ to an interpretation $\underline{\partial}'$ refining $\underline{\partial}$.

In essence, RefineTernary implements one step of the ternary CTL model-checking algorithm on PKSs [12]. Interpretations on subformulas are refined in a bottom-up fashion along the abstract syntax tree of the formula. This is done as in classical CTL model checking [18] but on ternary interpretations instead of binary ones. The extension toward ternary interpretations is rather straight forward for local formulas as well as $\exists\mathsf{X}$ formulas. To achieve ternary deduction for $\exists\mathsf{G}$ and $\exists\,\mathsf{U}$ formulas, an optimistic and a pessimistic run of the classical CTL deduction step is performed on binary interpretations of subformulas. To match our extension of the ternary semantics [12] toward the compositional setting of RSMs, our deduction depends on contexts $\underline{\gamma}$ the components are evaluated in. In the optimistic run all direct subformulas that are "unknown" are assumed to hold, while in the pessimistic run they are assumed to not hold. Hence, in the initialization the pessimistic run collects all nodes in which the relevant subformula (i.e., ϕ_1 for $\exists\mathsf{G}\Phi_1$ and ϕ_2 for $\exists\phi_1\,\mathsf{U}\,\phi_2$) surely holds, checking whether the interpretation $\underline{\partial}$ maps the formula to **tt** in a specific node, whereas the optimistic run collects all nodes in the subformula may hold, i.e., including nodes in which $\underline{\partial}$ maps the relevant subformula to **??** as well. Afterward, we use the standard CTL model checking scheme, utilizing a backward reachability analysis where again optimistic and pessimistic satisfaction for the subformulas is assumed in the respective run. 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.

Notice that RefineTernary only operates at the level of a single formula Φ and the optimistic and pessimistic runs are only required for $\exists\mathsf{G}$ and $\exists\,\mathsf{U}$ formulas. In these cases, the used backward reachability analysis only considers transitions in $\longrightarrow_+$, i.e., local transitions within a component and transitions from call nodes to entry nodes, however, not transitions from exit to return nodes. This is because for a call node (b, en) we surely know its successor is en. In contrast, for a return node (b, ex) we only know its predecessor was ex. While this does not mean that ex always has (b, ex) as a successor, it potentially has some other return node (b', ex). Thus, we cannot conclude with certainty whether (b, ex) is reachable from ex.

Thus, since dealing with $\exists\mathsf{G}$ and $\exists\,\mathsf{U}$ can be done locally, it is not necessary to transform CTL formulas before applying the algorithm. To then compute an interpretation of all sub-

Algorithm 1: REFINETERNARY($\mathcal{A}$, Φ, $\underline{\partial}$, $\underline{\gamma}$)

input : an RSM $\mathcal{A}$, a CTL formula Φ over AP, a ternary interpretation $\underline{\partial}\colon \mathsf{N}^{all} \times Subf(\Phi) \to \{\mathbf{tt}, \mathbf{ff}, \mathbf{??}\}$, and contexts $\underline{\gamma}$
output: a ternary interpretation $\underline{\partial}'\colon \mathsf{N}^{all} \times Subf(\Phi) \to \{\mathbf{tt}, \mathbf{ff}, \mathbf{??}\}$ following the ternary semantics

$Sat_{opt}, Sat_{pes} := \emptyset, \emptyset$
forall the $n \in \mathsf{N}^{all}$ *and* $\phi \in Subf(\Phi) \setminus \{\Phi\}$ **do** $\underline{\partial}'(n, \phi) = \underline{\partial}(n, \phi)$
forall the $n \in \mathsf{N}^{all}$ **do** $\underline{\partial}'(n, \phi) = \mathbf{??}$
if $\Phi = a$ *with* $a \in AP$ **then**
  forall the $n \in \mathsf{N}^{all}$ **do**
    if $a \in L(n)$ **then** $\underline{\partial}'(n, \Phi) = \mathbf{tt}$**else** $\underline{\partial}'(n, \Phi) = \mathbf{tt}$
if $\Phi = \neg\phi_1$ **then**
  forall the $n \in \mathsf{N}^{all}$ **do** $\underline{\partial}'(n, \Phi) = comp(\underline{\partial}'(n, \phi_1))$
if $\Phi = \phi_1 \vee \phi_2$ **then**
  forall the $n \in \mathsf{N}^{all}$ **do** $\underline{\partial}'(n, \Phi) = \max(\underline{\partial}'(n, \phi_1), \underline{\partial}'(n, \phi_2))$
if $\Phi = \exists\mathsf{X}\phi_1$ **then**
  forall the $n \in (\mathsf{N}^{all} \setminus Ex)$ **do** $\underline{\partial}'(n, \Phi) = \max_{n \longrightarrow_+ n'} \underline{\partial}'(n', \phi_1)$
  forall the $ex \in Ex$ **do** $\underline{\partial}'(ex, \phi) = \underline{\gamma}(ex, \phi)$
if $\Phi = \exists\mathsf{G}\phi_1$ **then**
  /* pessimistic run */
  $Sat_{pes} := \{n \in \mathsf{N}^{all} \setminus Ex \mid \underline{\partial}'(n, \phi_1) = \mathbf{tt}\} \cup \{n \in Ex \mid \underline{\gamma}(n, \Phi) = \mathbf{tt}\}$
  while *there is* $n \in Sat_{pes} \setminus Ex$ *with* $n' \notin Sat_{pes}$ *for all* $n \longrightarrow_+ n'$ **do**
    $Sat_{pes} := Sat_{pes} \setminus \{n\}$
  forall the $n \in Sat_{pes}$ **do** $\underline{\partial}'(n, \Phi) := \mathbf{tt}$
  /* optimistic run */
  $Sat_{opt} := \{n \in \mathsf{N}^{all} \setminus Ex \mid \underline{\partial}'(n, \phi_1) \neq \mathbf{ff}\} \cup \{n \in Ex \mid \underline{\gamma}(n, \Phi) \neq \mathbf{ff}\}$
  while *there is* $n \in Sat_{opt} \setminus Ex$ *with* $n' \notin Sat_{opt}$ *for all* $n \longrightarrow_+ n'$ **do**
    $Sat_{opt} := Sat_{opt} \setminus \{n\}$
  forall the $n \in \mathsf{N}^{all} \setminus Sat_{opt}$ **do** $\underline{\partial}'(n, \Phi) := \mathbf{ff}$
if $\Phi = \exists\phi_1 \mathsf{U} \phi_2$ **then**
  /* pessimistic run */
  $Sat_{pes} := \{n \in \mathsf{N}^{all} \setminus Ex \mid \underline{\partial}'(n, \phi_2) = \mathbf{tt}\} \cup \{n \in Ex \mid \underline{\gamma}(n, \Phi) = \mathbf{tt}\}$
  while *there is* $n \longrightarrow_+ n'$ *where* $n \notin Sat_{pes}, n' \in Sat_{pes}$ *and* $\underline{\partial}'(n, \phi_1) = \mathbf{tt}$ **do**
    $Sat_{pes} := Sat_{pes} \cup n$
  forall the $n \in Sat_{pes}$ **do** $\underline{\partial}'(n, \Phi) := \mathbf{tt}$
  /* optimistic run */
  $Sat_{opt} := \{n \in \mathsf{N}^{all} \setminus Ex \mid \underline{\partial}'(n, \phi_2) \neq \mathbf{ff}\} \cup \{n \in Ex \mid \underline{\gamma}(n, \Phi) \neq \mathbf{ff}\}$
  while *there is* $n \longrightarrow_+ n'$ *where* $n \notin Sat_{opt}, n' \in Sat_{opt}$ *and* $\underline{\partial}'(n, \phi_1) \neq \mathbf{ff}$ **do**
    $Sat_{opt} := Sat_{opt} \cup n$
  forall the $n \in \mathsf{N}^{all} \setminus Sat_{opt}$ **do** $\underline{\partial}'(n, \Phi) := \mathbf{ff}$
return $\underline{\partial}'$

formulas of Φ, we can simply call REFINETERNARY on all subformulas of Φ bottom-up as usual in CTL model checking. An extension of optimistic and pessimistic run toward other logics such as LTL fragments would require taking care of additional details. For instance, dealing with negation might require specific techniques, e.g., ensuring formulas in negation normal form [3, 7, 12, 43]. Differently, our algorithm does not face such particularities, since our bottom-up procedure treats every subformula in isolation.

On the level of the single formula Φ we indeed check the entire system nodes whether Φ holds under the ternary semantics. However, recall that the system representation as an RSM is always finite as opposed to the underlying Kripke structure, and even if the underlying Kripke structure is finite, it may be exponentially larger than the RSM. Thus, utilizing the compositional structure of the RSM potentially can make the ternary model-checking approach significantly faster than flattening the RSM and model-checking the resulting Kripke structure.

Example 6 Consider again the RSM in Example 3 and CTL formula $\Phi = \exists\mathsf{F}\exists\mathsf{G}\,\bigcirc = \exists\mathbf{tt}\,\mathsf{U}\,\exists\mathsf{G}\,\bigcirc$. Let us run REFINETERNARY on all subformulas of Φ, bottom-up. Assume that initially $\underline{\partial}$ maps all node-formula pairs to **??**. Also, assume that $\underline{\gamma}(n_4, \exists\mathsf{G}\,\bigcirc) = \mathbf{ff}$ and $\underline{\gamma}(n_6, \Phi) = \underline{\gamma}(n_6, \exists\mathsf{G}\bigcirc) = \mathbf{tt}$ and otherwise $\underline{\gamma}(\cdot, \cdot) = \mathbf{ff}$.

First, we run REFINETERNARY on $\bigcirc$. This trivially constructs $\underline{\partial}'$ such that it maps nodes to **tt** and **ff**, depending on their labels.

Next, we run the algorithm on $\exists\mathsf{G}\bigcirc$ and $\underline{\partial}'$. To extend $\underline{\partial}'$, we first copy $\underline{\gamma}$ for all inputs for which the output is

not **??**. So here, $\underline{\partial}'(n_6, \exists\mathsf{G}\bigcirc) = \mathbf{tt}$ and $\underline{\partial}'(n_4, \exists\mathsf{G}\bigcirc) = \mathbf{ff}$. For the pessimistic run, we follow the standard CTL model checking procedure for $\exists\mathsf{G}$ formulas and first collect all nodes in which $\bigcirc$ holds in the set Sat_{pes} and then successively remove all non-exit nodes which have no successor in Sat_{pes} until we reach a fixed point. This leaves us with $Sat_{pes} = \{n_6, n_3, (b_2, n_3), n_2, (b_1, n_6)\}$, in which $\exists\mathsf{G}\bigcirc$ definitely holds. For the optimistic run, collect all nodes in which $\bigcirc$ may hold in the set Sat_{opt}. Note that this includes nodes in which $\bigcirc$ would be **??**. However, since $\bigcirc$ is an atomic proposition, it is known in this example and thus this step is identical to the pessimistic run. We then successively remove all non-exit nodes without a Sat_{opt} successor. This results in $Sat_{opt} = \{n_6, n_3, (b_2, n_3), n_2, (b_1, n_6), n_5\}$ as a fixed point. For all other nodes $n \notin Sat_{opt}$ we thus set $\underline{\partial}'_i(n, \exists\mathsf{G}\bigcirc) = \mathbf{ff}$.

To refine $\underline{\partial}'$ for Φ we again perform a pessimistic and an optimistic run. The pessimistic run is again similar to the standard (binary) CTL model checking, additionally incorporating contextual information. We first collect all nodes in which Φ is known to hold, either a priori (e.g., through the context) or by $\exists\mathsf{G}\bigcirc$ holding. This is $\{n_6, n_3, (b_2, n_3), n_2, (b_2, n_6)\}$ in this example. We then collect all nodes than can reach a node in this set through backward reachability analysis, leading to Sat_{pes} containing all those node as well as n_1 in which Φ is now known to hold. For the optimistic run we again start by collecting nodes in which Φ may hold., optimistically assuming that $\exists\mathsf{G}\bigcirc$ holds in nodes where it is unknown, i.e., all nodes except n_1 and (b_1, n_4). Notice that Φ does optimistically hold in n_4 itself due to $\gamma_1(n_4, \Phi) = \mathbf{??}$. After reachability analysis, Sat_{opt} contains all nodes, meaning we do not set $\underline{\partial}'(\cdot, \Phi) = \mathbf{ff}$ for any node.

All node-formula pairs for which we have not set a truth value here agree with $\underline{\partial}$ and thus are mapped to **??**.

Lemma 3 *Let $\underline{\mathcal{A}} = (\mathcal{A}_1, \ldots, \mathcal{A}_k)$ be an RSM $\underline{\mathcal{A}}$ with consistent contexts $\underline{\gamma}$ over a CTL formula Φ. Further, let $\underline{\partial}$ be the ternary interpretation induce by the ternary semantics for all $n \in \mathsf{N}^{all}$ and $\phi \in Subf(\Phi)\backslash\{\Phi\}$, i.e., $\underline{\partial}(n, \phi) = [n \models \phi]_{\underline{\gamma}}$*

Then Algorithm 1 terminates and returns a ternary interpretation $\underline{\partial}'$ that follows the ternary semantics for all $n \in \mathsf{N}^{all}$ and $\phi \in Subf(\Phi)$, i.e., $\underline{\partial}'(n, \phi) = [n \models \phi]_{\underline{\gamma}}$.

5.3 Contextualization

The main difference of our REFINETERNARY method compared to a single deduction step by the standard ternary CTL model-checking algorithm [12] is that we explicitly give a consistent partial interpretation $\underline{\partial}$ and consistent context $\underline{\gamma}$ as input parameter. To this end, we can include assumptions on the satisfaction of subformulas in the deduction process such as knowledge on the environment a component is executed in, i.e., the box stack that lead to the component being executed. What remained open thus far is how we actually come up with a suitable interpretation and contexts as input for REFINETERNARY.

The issue is that if a box b invoking $\mathcal{A}_i$ is called multiple times, either at different locations or on different recursion levels, it is possible that all fully specified contexts, i.e., contexts that do not map any inputs to **??**, are inconsistent. Thus, Lemma 2 is not applicable and we may not directly obtain a result on whether a certain CTL formula is satisfied.

Example 7 Consider again the RSM in Example 3, and the CTL property $\Phi = \exists\mathsf{F}\exists\mathsf{G}\bullet$. Any context γ_1 for which $\gamma_1, (n_4\Phi) = \mathbf{ff}$ is inconsistent since $(\epsilon, n_4) \models \Phi$. However, also any context γ_1 for which $\gamma_1(n_4, \Phi) = \mathbf{tt}$ is inconsistent since $(b_1, n_4) \not\models \Phi$. Without this contextual knowledge, it is impossible to deduce whether $\underline{\mathcal{A}} \models \Phi$ using the ternary semantics, i.e., running REFINETERNARY.

To obtain complete results but still be able to reason about components in a modular way, we consider each component multiple times under different contexts. This is achieved by computing all *reachable* contexts and reasoning about each component separately for each context. To implement this, we introduce the function CONTEXTUALIZE, described in Algorithm 2, which maps $\underline{\mathcal{A}}$, a CTL formula Φ, contexts $\underline{\gamma}$, an interpretation $\underline{\partial}$ over $\underline{\mathcal{A}}$ and a target box $b \in B_i$ to a possibly modified RSM $\underline{\mathcal{A}}'$, in which the context of the component called by b is refined, potentially by creating a new component. Additionally, the function returns consistent contexts $\underline{\gamma}'$ and a consistent ternary interpretation $\underline{\partial}'$, both over $\underline{\mathcal{A}}'$ and Φ, The refinement of the context of box b can be formalized by requiring that the $\mathcal{A}'$ satisfies the following requirement for all $ex \in Ex_Y(b)$ and $\phi \in Subf_{\exists}(\Phi)$:

$$\underline{\gamma}'(ex, \phi) = \underline{\partial}((b, ex), \phi)$$

The challenge then becomes to modify the RSM in such a way that ensures that $\underline{\gamma}'$ is still consistent.

Our algorithm for CONTEXTUALIZE first checks whether there already exists a component $\mathcal{A}_j$ with context γ_j being equivalent to the context induced by b's return nodes. Formally, equivalence here means that $\mathcal{A}_j$ and $\mathcal{A}_{Y(b)}$ must be isomorphic, i.e., there exists an isomorphism f between N_j^{all} and $\mathsf{N}_{Y(b)}^{all}$ such that the transition relation $\longrightarrow$ is preserved, as well as an isomorphism g between boxes B_k and $B_{Y(b)}$ such that the referenced components, i.e., $\mathcal{A}_Y(b)$ and $\mathcal{A}_Y\big(g(b)\big)$ are the same. The contexts γ_j and γ_b are then equivalent if $\gamma_j(ex) = \gamma_b\big(f(ex)\big)$ for all $ex \in Ex_j$. On a technical level, to check whether such a context exists, whenever we create a copy of a component we track from which component was originally copied from, which we call a *base component*. If two components were copied from the same base component, they are always isomorphic. Thus, to check whether a context equivalent to γ_b exists, we only need to check whether

Algorithm 2: CONTEXTUALIZE($\underline{\mathcal{A}}$, Φ, $\underline{\gamma}$, $\underline{\partial}$, b)

input : an RSM $\underline{\mathcal{A}} = (\mathcal{A}_1, \ldots, \mathcal{A}_k)$, an existentially quantified CTL formula Φ, context $\underline{\gamma}$, ternary interpretation $\underline{\partial}$ over $\underline{\mathcal{A}}$, and a box $b \in B_i$
output: a modified RSM $\underline{\mathcal{A}}'$ with refined contexts $\underline{\gamma}'$ and interpretations $\underline{\partial}'$ for component $Y(b)$

```
forall the φ ∈ Subf(Φ) and (b, ex) ∈ Return_b do  γ_b(ex, φ) := ∂((b, ex), φ)
∂' := ∂
if there is j where γ_b and γ_j are equivalent then
    A' := A
    Y'_i(b) := j
else
    A_{k+1} := copy(A_{Y_i(b)})
    /* copies of nodes and boxes are denoted by ' */
    A' := (A_1, ..., A_k, A_{k+1})
    γ' := (γ_1, ..., γ_k, γ_b)
    forall the n' ∈ N^{all}_{k+1} and φ ∈ Subf(Φ) do  ∂'(n', φ) = ∂(n, φ)
    Y'_i(b) := k + 1
A' = RemoveUnreachableComponents(A')
return A', γ', ∂'
```

there is a component that has the same base component as $\mathcal{A}_Y(b)$ and for which γ_b agrees with its context on all inputs. If an equivalent context exists, we (re)assign b to the found contextualized component. Otherwise, a copy[2] $\mathcal{A}_{k+1}$ of the component $\mathcal{A}_{Y_i(b)}$ is generated (i.e., the number of components of the RSM increases from k to $k+1$) with context γ_b and the box b is reassigned to the fresh component $\mathcal{A}_{k+1}$ by updating function Y_i (see Sect. 3.1). As this new component structurally is a copy of $\mathcal{A}_{Y(b)}$, it is isomorphic to $\mathcal{A}_Y(b)$ and has the same base component. Lastly, we remove any components that became unreachable during contextualization which can be achieved by a simple graph reachability analysis. Note that this also means we remove its corresponding context from $\underline{\gamma}$.

Example 8 Consider again the RSM in Example 3 and CTL formula $\Phi = \exists\mathsf{F}\bullet$. Assume contexts $\underline{\gamma}(n_4, \Phi) = \mathbf{tt}$ and else $\underline{\gamma}(\cdot, \Phi) = \mathbf{??}$. Further, let ∂ be the result of running REFINETERNARY, i.e., $\partial(n_4, \bullet) = \partial((b_1 n_4), \bullet) = \mathbf{tt}$ and $\mathbf{ff}$ otherwise, and $\partial(n, \Phi)$ is $\mathbf{tt}$ for nodes $n \in \{n_4, (b_1, n_5)), (b_2, n_6)\}$ and otherwise $\mathbf{??}$.

Let us contextualize box b_1. The induced context follows ∂, i.e., we construct $\gamma_{b_1}(n_4, \Phi) = \gamma_{b_1}(n_5, \Phi) = \mathbf{tt}$. As this does not match the context γ_1 of $\mathcal{A}_1$ we construct a new component $\mathcal{A}_3$ and setting $Y_b = 3$, resulting in the RSM depicted in Fig. 5. Here, the ternary value behind each exit node ex specifies the value $\underline{\gamma}(ex, \Phi)$.

We can now extend ∂ toward including $\mathcal{A}_3$ by running REFINETERNARY again. Afterward, we can continue contextualizing, e.g. w.r.t. b_1' for which we again first construct the induced context $\gamma_{b_1'}(n_4, \Phi) = \gamma_{b_1'}(n_5, \Phi) = \mathbf{tt}$. In this case now a component exists which is isomorphic to $\mathcal{A}_{Y(b_1')} = \mathcal{A}_1$

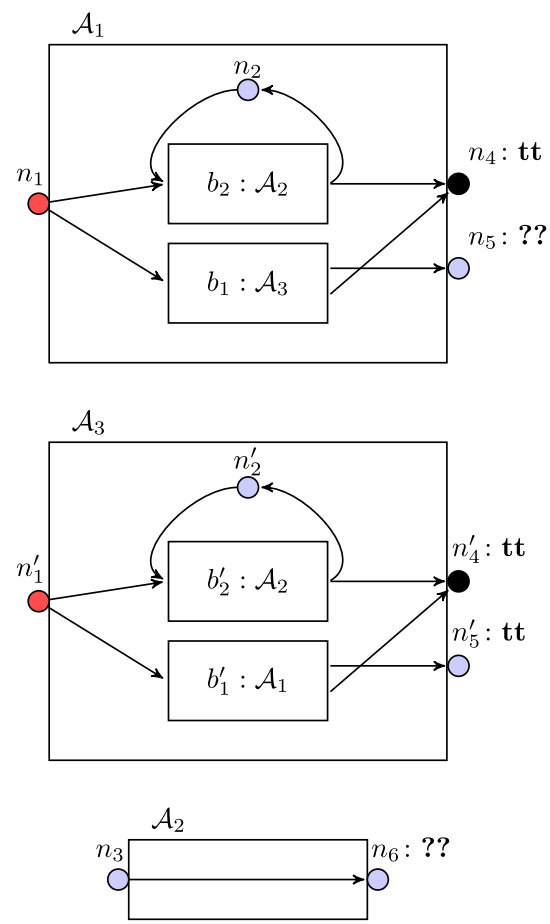


Fig. 5 Example RSM contextualized w.r.t. b_1 and $\Phi = \exists\mathsf{F}\bullet$

for which the context agrees with $\gamma_{b_1'}$, namely $\mathcal{A}_3$ itself. Thus, to refine b's context, we simply set $Y(b_1') = 3$.

Lemma 4 *Let $\underline{\mathcal{A}}$ be an RSM with consistent contexts $\underline{\gamma}$ and consistent interpretation $\underline{\partial}$ over a CTL formula Φ. Then for*

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

any input box b Algorithm 2 returns $\underline{\gamma}'$ and $\underline{\partial}'$ that are consistent with $\underline{\mathcal{A}}'$.

Algorithm 3: INITIALIZE($\underline{\mathcal{A}}$, Φ)

input : an RSM $\underline{\mathcal{A}} = (\mathcal{A}_1, \ldots, \mathcal{A}_k)$ and a CTL formula, both Φ over AP
output: initial RSM $\underline{\mathcal{A}}$ with contexts $\underline{\gamma}$

1 **forall the** $i \in \{1, \ldots, k\}$ **do**
2 | **forall the** $n \in \mathsf{N}_i^{all}$ *and* $\phi \in [Subf(\Phi)]$ **do** $\gamma_i(n, \phi) =$ **??**
3 $\mathcal{A}_0 = copy(\mathcal{A}_1)$
4 **forall the** $\phi \in [Subf(\Phi)]$ **do**
5 | **forall the** $ex \in Ex_0$ **do**
6 | | **if** $\phi = a$ *for* $a \in AP$ **then**
7 | | | $\partial_{init}(ex, \phi) := L_0(ex, a)$
8 | | **if** $\phi = \neg\phi_1$ **then**
9 | | | $\partial_{init}(ex, \phi) := \neg\partial_{init}(ex, \phi_1)$
10 | | **if** $\phi = \phi_1 \vee \phi_2$ **then**
11 | | | $\partial_{init}(ex, \phi) := \partial_{init}(ex, \phi_1) \vee \partial_{init}(ex, \phi_2)$
12 | | **if** $\phi = \exists\mathsf{G}\phi_1$ *or* $\phi = \exists\mathsf{X}\phi_1$ **then**
13 | | | $\partial_{init}(ex, \phi) := \partial_{init}(ex, \phi_1)$
14 | | | $\gamma_0(ex, \phi) := \partial_{init}(ex, \phi)$
15 | | **if** $\phi = \exists\phi_1 \mathsf{U} \phi_2$ **then**
16 | | | $\partial_{init}(ex, \phi) := \partial_{init}(ex, \phi_2)$
17 | | | $\gamma_0(ex, \phi) := \partial_{init}(ex, \phi)$
18 $\underline{\mathcal{A}}' = (\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_k)$
19 $\underline{\gamma}' := \gamma_0 \cup \bigcup_{i=1}^{k} \gamma_i$
20 **return** $\underline{\mathcal{A}}', \underline{\gamma}$

5.3.1 Initial context

To obtain a complete result, it is not sufficient to alternate between calling REFINETERNARY and CONTEXTUALIZE. One issue that is left, is that we have not handled the Kripke structure semantics rule (loop) yet (cf. Fig. 2). Recall that this rule specifies that that if we reach an exit node ex with the empty box stack ϵ, we loop forever in ex. In order to properly capture this behavior, we explicitly handle this case by first contextualizing the outermost component as this is the only component that we can be in while the box stack is empty.

Formally, we construct a new initial component $\mathcal{A}_0$ with the same structure as the original initial component along with a context γ_0 over its exit nodes Ex_0 and all existential subformulas $\phi \in Subf_{\exists}(\Phi)$ such that

$$\gamma_0(ex, \phi) = \mathbf{tt} \iff (\epsilon, ex) \models \phi$$
$$\gamma_0(ex, \phi) = \mathbf{ff} \iff (\epsilon, ex) \not\models \phi$$

To contextualize the outermost component, which does not depend on a calling component, we utilize a function INITIALIZE that is implemented by Algorithm 3.

The intuition behind the algorithm is rather straight forward. First, we initialize all contexts to **??** in Line 2. Then, in Line 3 we create a copy of the initial component $\mathcal{A}_0$ for which we aim to construct a ternary interpretation ∂_{init} over its exit nodes Ex_0 inducing an initial context that it is consistent with the rule (loop) of the underlying Kripke structure $[\![\underline{\mathcal{A}}]\!]$. The reason for creating a copy is that we do only know the context of $\mathcal{A}_0$ under the empty box stack ϵ. However, $\mathcal{A}_1$ may also be invoked by other boxes and thus in a different context.

To construct the maximally consistent interpretation ∂_{init} over all nodes in Ex_0, we perform local deduction steps for all $\phi \in Subf(\Phi)$ in a bottom-up fashion as for standard CTL model checking. We will denote this ordered selection of subformulas ϕ, where all subformulas of ϕ already have been selected before, by $\phi \in [Subf(\Phi)]$ in the following. For the propositional formulas (cf. Lines 6–11) the deduction is straight forward. For existentially quantified formulas (cf. Line 12 to Line 17), since the only path from an exit node ex with an empty stack is a self-loop, we know that $\exists\mathsf{G}\phi_1$ and $\exists\mathsf{X}\phi_1$ hold in ex iff ϕ_1 holds in ex, and $\phi = \exists\phi_1 \mathsf{U} \phi_2$ holds in ex iff ϕ_2. The initial context γ_0 then is the projection of ∂_{init} on nodes in $Subf_{\exists}(\Phi)$. Notice that due this recursive definition, the initial context is fully specified, i.e., never returns **??** for any node-formula pair.

Finally, we add the new component along with its context to the RSM and return it.

Example 9 Consider again the RSM in Example 3 and CTL formula $\Phi = \exists\mathsf{F}\exists\mathsf{G}\bullet$.

For the initialization, we create a new initial component $\mathcal{A}_0$ in which we check whether which formulas hold in which exit nodes. We denote nodes in $\mathcal{A}_0$ with a superscript 0. In this case, all subformulas of Φ are satisfied in n_4^0 whereas none are satisfied n_5'. Thus, γ_{init} maps (n_4^0, Φ) and $(n_4^0, \exists\mathsf{G}\bullet)$ to **tt**, and(n_5', Φ) and $(n_5^0, \exists\mathsf{G}\bullet)$ to **ff**.

Note that all boxes refer to either $\mathcal{A}_1$ or $\mathcal{A}_2$, but not to the initial component $\mathcal{A}_0$, ensuring that $\mathcal{A}_0$ is only reached with the empty box stack ϵ. However, it is possible that later in the model checking procedure CONTEXTUALIZE modifies the RSM by redefining $Y(b)$ for some box b such that b is referencing $\mathcal{A}_0$. In that case $\mathcal{A}_0$ can then also be reached with some non-empty box stack. This is not a problem since as we have seen CONTEXTUALIZE ensures that this redefining maintains consistency of $\underline{\gamma}$.

Lemma 5 *For an RMS $\underline{\mathcal{A}}$ and CTL formula Φ Algorithm 3 returns an RSM $\underline{\mathcal{A}}'$ with consistent contexts $\underline{\gamma}'$.*

5.4 Eager RSM model checking

Piecing together the algorithms sketched so far, we devise 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 toward a global satisfaction relation.

Algorithm 4: EAGERCHECK($\underline{\mathcal{A}}$, Φ)

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

1 $\underline{\mathcal{A}}', \underline{\gamma}' :=$ INITIALIZE($\underline{\mathcal{A}}$, Φ)
2 **forall the** $\phi \in [Subf(\Phi)]$ **do**
3 **repeat**
  /* exhaustively construct reachable contexts */
4 **if** Φ *is existentially quantified* **then**
5 **forall the** $b \in B$ **do** $\underline{\mathcal{A}}, \underline{\partial} :=$ CONTEXTUALIZE($\underline{\mathcal{A}}, \phi, \underline{\gamma}, \underline{\partial}, b$)
6 $\underline{\partial} :=$ REFINETERNARY($\underline{\mathcal{A}}, \phi, \underline{\partial}$)
7 **until** $\underline{\partial}$ *did not change*
8 **forall the** $n \in \mathsf{N}^{all}$ *with* $\underline{\partial}(ex, \phi) = ??$ **do**
9 **if** $\phi = \exists \mathsf{G} \psi$ **then**
10 $\underline{\partial}(ex, \phi) :=$ **tt**
11 **if** $n \in Ex$ **then** $\underline{\gamma}(n, \phi) :=$ **tt**
12 **if** $\phi = \exists \psi_1 \mathsf{U} \psi_2$ **then**
13 $\underline{\partial}(ex, \phi) :=$ **ff**
14 **if** $n \in Ex$ **then** $\underline{\gamma}(n, \phi) :=$ **tt**
15 **if** *there is* $en \in En_1$ *with* $\underline{\partial}(en, \Phi) = ff$ **then return ff**
16 **else return tt**

The procedure follows ideas by Alur et al. [3] 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 4 shows the decision procedure EAGERCHECK($\underline{\mathcal{A}}$, Φ) that decides for an RSM $\underline{\mathcal{A}}$ and a CTL formula Φ whether $\underline{\mathcal{A}} \models \Phi$ holds or not. The algorithm starts with an initialization of the local ternary interpretations of the components of $\underline{\mathcal{A}}$ (function INITIALIZE) in Line 1. After initialization, in Line 2 EAGERCHECK 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 (cf. Line 5) and a ternary deduction by REFINETERNARY, refining local interpretations of components and determining new contexts toward a propagation from calling components to called ones (cf. Line 6). This is done until we reach a fixed point, i.e., the ternary interpretation is not refined any further by REFINETERNARY and CONTEXTUALIZE can no longer refine any contexts.

5.4.1 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., $\underline{\partial}$ may still map to **??**. Intuitively, this can happen when the context of a box depends on the evaluation of the boxes' entry nodes. For this, we need to have a cycle that passes through an exit node and thus hinders refinement through contextualization.

Formally, we define a *global dependency cycle* in an RSM $\underline{\mathcal{A}}$ w.r.t. a formula $\Phi = \exists \mathsf{G} \phi_1$ or $\Phi = \exists \phi_1 \mathsf{U} \phi_2$ as a cyclic sequence of nodes $n_1 n_2 \ldots n_\ell n_1$ with

- $n_i \in \mathsf{N}^{all}$ for all $i \in \{1, \ldots, \ell\}$
- $n_j \in Ex$ for at least one $j \in \{1, \ldots, \ell\}$
- $(\sigma, n_i) \models \phi_1$ for all $\sigma \in B^*$ and $i \in \{1, \ldots, \ell\}$
- in the $\exists \mathsf{U}$-case $(\sigma, n_i) \not\models \phi_2$ for all $\sigma \in B^*$ and $i \in \{1, \ldots, \ell\}$

such that there exists a cyclic path $(\cdot, n_1) \Longrightarrow (\cdot, n_2) \Longrightarrow \ldots \Longrightarrow (\cdot, n_\ell) \Longrightarrow (\cdot, n_1)$ in the underlying Kripke structure $[\![\underline{\mathcal{A}}]\!]$ and there is no witnessing path for Φ, except potentially such a cycle. We illustrate this situation on our running example:

Example 10 Consider again the RSM in Example 3 and CTL formula $\Phi = \exists \mathsf{G} \bigcirc$.

To initialize, we create a component $\mathcal{A}_0$ and obtain the RSM $\underline{\mathcal{A}}$ with context $\underline{\gamma}$ that maps $(\cdot, \Phi)$ to **tt** for n_5^0, to **ff** for n_4^0 and to **??** for all other exit nodes. Running REFINETERNARY we find that Φ does not hold in $n_1, (b_1, n_1), n_4, (b_1, n_4), (b_1, n_5)$ or any of their copies. Contextualizing w.r.t. all boxes yields that neither b_2 nor b_2^0 can be refined as $(b_2, n6) = (b_2', n_6) = ??$. For b_1' and b_1^0 we see that both induced contexts $\gamma_{b_1'}$ and $\gamma_{b_1^0}$ map every node-formula pair to false since we found that neither (b_1, n_4) nor (b_1, n_5) or any of their copies satisfy Φ. Thus, we create a new component $\mathcal{A}_3$ by copying $\mathcal{A}_1$ and assigning it context $\gamma_{b_1'}$, and redirect b_1' and b_1^0 to $\mathcal{A}_3$. This makes $\mathcal{A}_1$ unreachable; hence, we remove it. This leaves us in the situation depicted in Fig. 6. Here, the truth value behind each node n represents $\underline{\partial}(n, \Phi)$. We can see that Φ is still unknown in

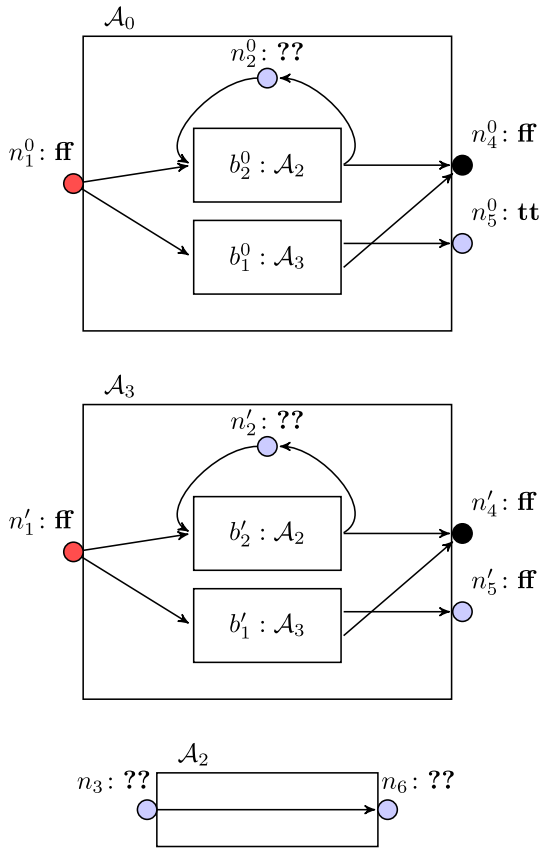


Fig. 6 Example RSM contextualized w.r.t. b_1 and $\Phi = \exists \mathsf{G} \circ$

n_2, n_3 and n_6 as well as in all corresponding box call and return nodes (b_2, n_3), (b_2, n_6) and all copies of them. We cannot refine any contexts via CONTEXTUALIZE since for all boxes b the induced context γ_b is the same as the context of the referenced component $\gamma_{Y(b)}$. Also REFINETERNARY cannot refine $\underline{\partial}$ any further since determining Φ in, e.g., (b_2, n_6) would requires knowing $\gamma_2(n_6)$ which in turn depends on the value of Φ in (b_2, n_6).

Intuitively, we thus have a cycle of dependencies connected through several components that hinders further refinement via CONTEXTUALIZE and REFINETERNARY. If such cycles appear locally, REFINETERNARY can take care of them, but in this case, where the cycle traverses an exit node, we cannot make progress via CONTEXTUALIZE and REFINETERNARY.

Since we found this cycle now, we can manually deduce that this is a witness for Φ holding in this cycle and set the context $\gamma_2(n_6, \Phi) = \mathbf{tt}$.

In general, we resolve such situations by the following reasoning: Since there is a dependency cycle that hindered refinement of $\Phi = \exists \mathsf{G} \phi_1$, all nodes on this cycle have to satisfy ϕ_1. Thus, this cycle can serve as a witness of Φ to hold and we refine all contexts for Φ and exit nodes on the cycle toward **tt**. A similar argumentation can be applied when $\Phi = \exists \phi_1 \mathsf{U} \phi_2$ formula but with refining all **??**-nodes toward **ff** since the ϕ_1 cycle hindering further refinement is not a witness for Φ, and other possible witnesses for Φ would have been found by alternating REFINETERNARY and CONTEXTUALIZE.

We can also show formally that the two scenarios described above is the only reason that a fixed point in which $\underline{\partial}(\cdot, \Phi) = \mathbf{??}$ can occur. In the next lemma, we prove that this can only happen for $\exists \mathsf{G}$ and $\exists \mathsf{U}$ formulas, and that for $\exists \mathsf{G}$ formulas this immediately yields a witness, while for $\exists \mathsf{U}$ formulas the formula does not hold on cycles.

Lemma 6 *Let $\underline{\mathcal{A}}$ be an RSM with consistent contexts $\underline{\gamma}$ and consistent interpretation $\underline{\partial}$ over a formula Φ. If $\underline{\mathcal{A}}$ has been initialized via* INITIALIZE, *and $\underline{\partial}$ =* REFINETERNARY *$(\underline{\mathcal{A}}, \Phi, \underline{\partial}, \underline{\gamma})$ and $\underline{\mathcal{A}}, \underline{\gamma}, \underline{\partial}$ =* CONTEXTUALIZE*$(\underline{\mathcal{A}}, \Phi, \underline{\gamma}, \underline{\partial}, b)$ for all $b \in B$, i.e., both are fixed points, and $\underline{\partial}(n, \phi) \neq$ **??** $\neq \underline{\gamma}(n, \phi)$ for all $n \in \mathsf{N}^{all}$ and $\phi \in Subf(\Phi) \backslash \{\Phi\}$, then $\underline{\partial}(n, \Phi) =$ **??** implies that either*

- *$\Phi = \exists \mathsf{G} \phi_1$ and $(\sigma, n) \models \Phi$ for all $\sigma \in B^*$, or*
- *$\Phi = \exists \phi_1 \mathsf{U} \phi_2$ and $(\sigma, n) \not\models \Phi$ for all $\sigma \in B^*$.*

This is reflected in Algorithm 4 in Line 8 to Line 14 where we modify $\underline{\partial}$ according to Lemma 6.

Note that our efficient resolution of global dependency cycles relies on ternary deduction, since cycles of unrefinable **??**-nodes directly provide information about the satisfaction of CTL formulas. While our algorithm is based on [3], 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.

5.4.2 Eager RSM model checking

Taking global dependency cycle resolution into account and piecing together the algorithms discussed so far, we obtain correctness of our eager model-checking algorithm EAGERCHECK:

Theorem 1 *Algorithm 4 terminates and is correct, i.e., returns **tt** iff $\underline{\mathcal{A}} \models \Phi$ and **ff** iff $\underline{\mathcal{A}} \not\models \Phi$ for any RSM $\underline{\mathcal{A}}$ and CTL formula Φ over a common set of atomic propositions.*

6 Lazy RSM model checking

The model-checking algorithm presented in Sect. 5 mainly combined existing techniques for model-checking RSMs with ternary model checking techniques for CTL formulas [3, 5, 12, 15, 18]. In this section, we reuse the elements of Algorithm 4 toward heuristics to reduce 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 toward lazy evaluation of subformulas and components, respectively.

6.1 Lazy contextualization

Eager 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 [9]. Reducing the number of contexts considered during the deduction process thus provides a potential to speed up the model-checking process.

6.1.1 Lazy formula evaluation

The main idea toward 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.

Example 11 Let us consider the RSM of Fig. 3 and $\Phi = \exists \mathsf{X} \bullet \vee \exists \mathsf{X}(\exists \circ \mathsf{U} \bullet)$. Then, satisfaction of Φ can be determined by solely regarding $\phi = \exists \mathsf{X} \bullet$ 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 (b_1, n_1) is labeled by $\bullet$ 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 $\underline{\mathcal{A}} \models \Phi$. In this example, we reduced the number of contexts to be evaluated as we did not evaluate any context for component $\mathcal{A}_2$.

6.1.2 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.

Given an RSM $\underline{\mathcal{A}}$ with contexts $\underline{\gamma}$ and ternary interpretation $\underline{\partial}$ over a CTL formula Φ, we define its *contextualizable* boxes $C(\mathcal{A}, \underline{\gamma}, \underline{\partial})$ as the set of boxes which contain more contextual information than their corresponding component, i.e.,

$$C(\mathcal{A}, \Phi, \underline{\gamma}, \underline{\partial}) = \big\{ b \in B \mid \text{there is } n \in \mathit{Ex}_{Y(b)} \text{ and } \phi \in \mathit{Subf}_{\exists}(\Phi) \text{ s.t. } \underline{\partial}\big((b,n),\phi\big) \neq \mathbf{??} \text{ and } \underline{\gamma}(n,\phi) = \mathbf{??} \big\}$$

An *expansion heuristic* H is any function that maps an RSM $\underline{\mathcal{A}}$ with contexts $\underline{\gamma}$ and ternary interpretation $\underline{\partial}$ over a CTL formula Φ to a subset of $C(\mathcal{A}, \Phi, \underline{\gamma}, \underline{\partial})$. Intuitively, an expansion heuristic returns boxes that are to be contextualized, including $\emptyset$ if no box shall be contextualized. We call an expansion heuristic H *complete* iff it returns $\emptyset$ only when there are no contextualizable boxes, i.e.,

$$H(\mathcal{A}, \Phi, \underline{\gamma}, \underline{\partial}) = \emptyset \iff C(\mathcal{A}, \Phi, \underline{\gamma}, \underline{\partial}) = \emptyset.$$

6.2 Lazy approach

The idea of lazy contextualization of boxes in an RSM can be incorporated into the eager RSM model-checking approach EagerCheck presented in Algorithm 4. This leads to a method LazyCheck presented in Algorithm 5. We outline the difference in the workflow of both approaches in Fig. 7.

One of the main differences is that the lazy approach no longer follows a strict bottom-up approach. While the eager approach only considers a formula $\phi \in \mathit{Subf}(\Phi)$ once the truth value for all strict subformulas of ϕ are known (cf. Line 2 in Algorithm 4), the lazy approach attempts to compute whether $\underline{\mathcal{A}} \models \Phi$ before knowing the value of all subformulas for Φ in all nodes. To employ such reasoning we heavily rely on the ternary model-checking approach. In fact, in Line 3 we already compute the maximally consistent ternary interpretation before contextualizing any boxes (apart from the initial context) and check in Line 4 whether that is sufficient to determine whether $\underline{\mathcal{A}} \models \Phi$. If the lazy approach cannot deduce whether $\underline{\mathcal{A}} \models \Phi$, it additionally performs contextualization in a lazy fashion. While EagerCheck surely contextualizes all boxes with contexts encountered during ternary deduction RefineTernary, Algorithm 5 calls an expansion heuristic H in Line 5 to find boxes B whose contextualization might contribute to deciding whether the target formula Φ holds in the outermost component of the RSM. Then it proceeds to contextualize $\underline{\mathcal{A}}$ with respect to Φ and the boxes in B in Line 16. As a special case, if the expansion heuristic does not find any boxes to contextualize, we do a global dependency cycle resolution and modify $\underline{\gamma}$ in Line 6 to Line 14 according to Line 6.

Due to consistency of our ternary reasoning implemented in RefineTernary and the progress and contextualizing boxes through H in combination a global dependency cycle resolution similar as described in Line 5.4.1, we obtain correctness and soundness of our new model-checking algorithm for RSMs for complete expansion heuristics H.

Theorem 2 *Algorithm 5 terminates and is correct for complete expansion heuristic H, i.e.,* LazyCheck($\underline{\mathcal{A}}, \Phi, H$)

Algorithm 5: LAZYCHECK($\underline{\mathcal{A}}$, Φ, H)

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

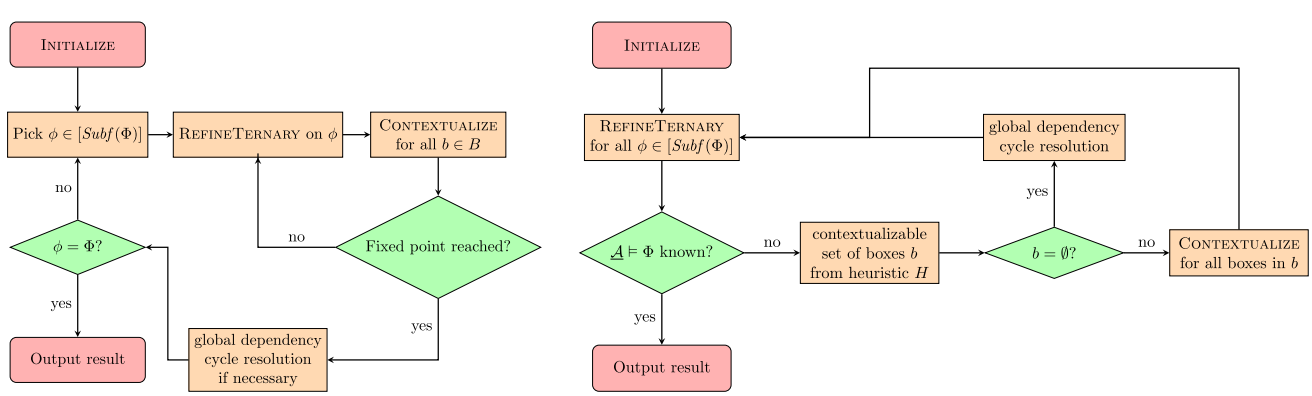
```
1  A', γ := INITIALIZE(A, Φ)
2  forall the n ∈ N^all and φ ∈ Subf(Φ) do ∂(n, φ) := ??
3  forall the φ ∈ [Subf(Φ)] do ∂ := REFINETERNARY(A, φ, ∂, γ)
4  while there is en ∈ En_1 with ∂(en, Φ) = ?? do
5      B = H(A, Φ, γ, ∂)
6      if B = ∅ then
7          forall the φ = ∃Gφ_1 ∈ Subf_∃(Φ) for which all ψ ∈ Subf(φ) \ {φ} are fully determined do
8              forall the n ∈ N^all with ∂(n, φ) = ?? do
9                  ∂(n, φ) = tt
10                 if n ∈ Ex then γ(n, φ) = tt
11         forall the φ = ∃φ_1 U φ_2 ∈ Subf_∃(Φ) for which all ψ ∈ Subf(φ) \ {φ} are fully determined do
12             forall the n ∈ N^all with ∂(n, φ) = ?? do
13                 ∂(n, φ) = ff
14                 if n ∈ Ex then γ(n, φ) = ff
15     else
16         forall the b ∈ B do CONTEXTUALIZE(A, Φ, γ, ∂, b)
17     forall the φ ∈ [Subf(Φ)] do
18         ∂ := REFINETERNARY(A, φ, ∂, γ)
19     if there is en ∈ En_1 with ∂(en, Φ) = ff then return ff
20 return tt
```

*returns **tt** iff $\underline{\mathcal{A}} \models \Phi$ and **ff** iff $\underline{\mathcal{A}} \not\models \Phi$ for any RSM $\underline{\mathcal{A}}$, and CTL formula Φ over a common set of atomic propositions.*

6.3 Incomplete expansion heuristics

So far we have considered only complete expansion heuristics. While they make use of lazy contextualization and ternary model checking, there are still plenty of scenarios where a complete expansion heuristic demands the contextualization of boxes that do not contribute to the final result. In particular, this affects scenarios where a global cycle check is necessary but which under a complete expansion heuristic can only be started when $C(\underline{\mathcal{A}}, \Phi, \underline{\gamma}, \underline{\partial}) = \emptyset$, i.e., when there are no more boxes to be contextualized.

Example 12 Consider the RSM given in Example 3 and the CTL formula $\Phi = \forall \mathsf{X} \exists \bigcirc \mathsf{U} \bullet$. To determine whether $\underline{\mathcal{A}} \models \Phi$ we have to check whether $\phi_1 = \exists \bigcirc \mathsf{U} \bullet$ holds in all successor nodes of n_1. It is immediately clear that (b_1, n_1) satisfies ϕ_1 as (b_1, n_1) has label $\bullet$. However, as discussed in the global cycle check paragraph of Example 10, determining whether ϕ holds in the nodes in the upper part of the RSM, in particular (b_2, n_3) requires a global dependency cycle resolution w.r.t. Φ. In LAZYCHECK with a complete expansion heuristic the global dependency cycle resolution can only be done once all contexts match the interpretation in the return nodes of all boxes, in particular $\gamma(n_4, \phi) = \underline{\partial}((b_1, n_4), \phi)$ and $\gamma(n_5, \phi) = \underline{\partial}((b_1, n_5), \phi)$ for all $\phi \in Subf(\Phi)$. However, after running REFINETERNARY we have that $\underline{\partial}((b_1, n_4), \phi) =$



(a) Eager workflow (b) Lazy workflow

Fig. 7 Workflows of the eager and lazy RSM model-checking approaches

Algorithm 6: GETNEXTEXPANSION($\underline{\mathcal{A}}$, Φ, $\underline{\gamma}$, $\underline{\partial}$)

input : RSM $\underline{\mathcal{A}} = (\mathcal{A}_1, \ldots, \mathcal{A}_k)$, a formula Φ, contexts $\underline{\gamma}$, and a ternary interpretation $\underline{\partial}$
output: a box b to contextualize, or a set of node-formula pairs D in a global dependency cycle

1 $D_{all} = \emptyset$
2 **forall the** $en \in En_1$ *with* $\underline{\partial}(en, \Phi) = ??$ **do**
3 $\quad b, D =$ FINDREASON($\underline{\mathcal{A}}$, en, Φ, $\underline{\gamma}$, $\underline{\partial}$, ϵ, $\emptyset$, $\emptyset$)
4 $\quad$ **if** $b \neq \emptyset$ **then return** b
5 $\quad D_{all} := D_{all} \cup D$
6 **return** D_{all}

$\underline{\partial}((b_1, n_5), \phi) = \mathbf{ff}$, whereas $\gamma(n_4, \phi) = \underline{\partial}(n_5, \phi) = ??$. Thus any complete expansion heuristic H by definition must return $\{b_1\}$ in this scenario even though it is clear that any contextualization of b_1 will not yield any new information as to whether $n_1 \models \Phi$ holds. Of course we can amplify this effect by directing b_1 to another, arbitrarily complex component and adding further conjunctions to Φ, such that LAZYCHECK requires exponentially many unnecessary contextualization steps.

6.3.1 Top-down expansion heuristic

Toward recognizing whether a contextualization potentially aids in verifying a property over an RSM, we provide an incomplete expansion heuristic by the function GETNEXTEXPANSION, specified by Algorithm 6 that calls the recursively defined function FINDREASON. The latter utilizes classical reasoning for local formulas and the well-known expansion laws of CTL for path formulas, in particular:

$$\exists \mathsf{G}\phi \equiv \phi \vee \exists \mathsf{X}\mathsf{G}\phi$$
$$\exists \phi_1 \,\mathsf{U}\, \phi_2 \equiv \phi_2 \vee \exists(\phi_1 \wedge \mathsf{X}\phi_1 \,\mathsf{U}\, \phi_2)$$

For this, FINDREASON traverses Φ in a top-down fashion to reason on *why* Φ is unknown in a node n and to find a box b where adding a subformula to its context might refine the interpretation of Φ in n. By only contextualizing heuristically selected boxes that may contribute to determining whether $\underline{\mathcal{A}} \models \Phi$ holds, we can potentially save contextualization steps.

Algorithm 7 considers several cases during recursion, from which we exemplify the most significant ones. In the first line we collect all node-formula pairs for which FINDREASON has been called. This is to later on prevent infinite cyclical calls of FINDREASON. First, those properties that could be locally resolved are considered. For instance, Line 3 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, the first disjunct ϕ_i for which $\underline{\partial}(n, \phi_i) = ??$ is chosen and FINDREASON is recursively called, determining a box b for which contextualization could resolve whether ϕ_i holds in n. If no such box exists, FINDREASON returns $\emptyset$ and the remaining disjuncts are checked. The cases of entering and leaving a box b are considered in Line 7 and Line 8, respectively. Notably, if n is an exit node, we consider the satisfaction of Φ in all possible calling components. If we find a box $b \in C(\mathcal{A}, \Phi, \underline{\gamma}, \underline{\partial})$, i.e., a box where contextualizing yields additional information, we return that box as our base case in Line 9. Otherwise, we continue our search in all possible return nodes (b, n). For existential path properties, let us exemplify the case where $\Phi = \exists \phi_1 \,\mathsf{U}\, \phi_2$ (see Line 26). Here, we determine the next recursive call arguments following the expansion law of $\exists \mathsf{U}$. First, we consider the local cases where ϕ_2 or ϕ_1 are unknown in n, asking for a box to contextualize by invoking FINDREASON on ϕ_2 and ϕ_1, respectively. Notice we check ϕ_2 first here, since the truth value $\exists \phi_1 \,\mathsf{U}\, \phi_2$ may solely be determined by the truth value ϕ_2 in some cases, but never by only ϕ_1's truth value. If both ϕ_1 and ϕ_2 are known but Φ is yet unknown in n, the reason for is not local in n and we continue in a successor node n' of n where Φ is still unknown. Similar to the disjunction case, we recursively call FINDREASON on all successors, returning any contextualizable box containing helpful information found, and returning $\emptyset$ if no such box was found after traversing all successors. Additionally, notice that we do not call FINDREASON recursively if we called it before already on the same node, signaled by the pair (n', Φ) being in R. As mentioned before, this is to prevent the algorithm calling itself in an infinite recurring cycle. Note that we only consider the node and formula for which FINDREASON was called to detect such behavior, but not the box stack, since the box stack merely serves as a tool for determining the successor of exit nodes here.

6.3.2 Modified global dependency cycle resolution

In the eager model checking algorithm (cf. Algorithm 4) alternating CONTEXTUALIZE and REFINETERNARY until a fixed point is reached does not guarantee that the ternary interpretation is maximally consistent due to potential global dependency cycles as shown in Example 10.

If FINDREASON enters a global dependency cycle, it eventually reaches a point where R contains all node-formula pairs on the dependency cycle and attempts to call itself recursively on a node-formula pair $(n', \Phi) \in R$ (see Line 21 and Line 30). If this is the case, we can surely determine that there is a cycle and also deduce which nodes lie on the cycle, namely all those that have been added to R since the last time FINDREASON was called on (n', Φ). On a technical level, this can easily be done by implementing R as an ordered list rather than a set. Importantly, R is used as a local variable here. This ensures that whenever FINDREASON is executed, R contains exactly the node-formula pairs that lead to the current call,

Algorithm 7: FINDREASON($\underline{\mathcal{A}}$, n, Φ, $\underline{\gamma}$, $\underline{\partial}$, R, D)

input : RSM $\underline{\mathcal{A}} = (\mathcal{A}_1, \ldots, \mathcal{A}_k)$, node $n \in N_i$, formula Φ, contexts $\underline{\gamma}$, a ternary interpretation $\underline{\partial}$, set of requested node-formulas pairs R, and a set of node-formula pairs in a global dependency cycle D
output: a singleton set of a box b to contextualize

1 $R := R \cup \{(n, \Phi)\}$
2 **if** $\Phi = \neg\phi_1$ **then return** FINDREASON($\underline{\mathcal{A}}$, n, ϕ_1, $\underline{\gamma}$, $\underline{\partial}$, R, D)
3 **if** $\Phi = \phi_1 \vee \ldots \vee \phi_\ell$ **then**
4 | **forall the** $j \in \{1, \ldots, \ell\}$ *with* $\underline{\partial}(n, \phi_j) = ??$ **do**
5 | | $b, D :=$ FINDREASON($\underline{\mathcal{A}}$, n, ϕ_j, $\underline{\gamma}$, $\underline{\partial}$, R, D)
6 | | **if** $b \neq \emptyset$ **then return** b, D
7 **if** $n = (b, en) \in Call_i$ **then return** FINDREASON($\underline{\mathcal{A}}$, en, Φ, $\underline{\partial}$, R, D)
8 **if** $n \in Ex$ *and there is* $b \in B$ *with* $\underline{\partial}((b, n), \Phi) \neq ??$ **then**
9 | **return** $\{b\}, D$ `// base case`
10 **else**
11 | **forall the** $b \in B$ *such that* $(b, n) \in \mathsf{N}^{all}$ **do**
12 | | $b, D :=$ FINDREASON($\underline{\mathcal{A}}$, (b, n), Φ, $\underline{\gamma}$, $\underline{\partial}$, R, D)
13 | | **if** $b \neq \emptyset$ **then return** b, D
14 **if** $\Phi = \exists \mathsf{X} \phi_1$ **then**
15 | **forall the** n' *with* $n \longrightarrow n'$ **do**
16 | | $b, D :=$ FINDREASON($\underline{\mathcal{A}}$, n', ϕ_1, $\underline{\gamma}$, $\underline{\partial}$, R, D)
17 | | **if** $b \neq \emptyset$ **then return** b, D
18 **if** $\Phi = \exists \mathsf{G} \phi_1$ **then**
19 | **if** $\underline{\partial}(n, \phi_1) = ??$ **then return** FINDREASON($\underline{\mathcal{A}}$, n, ϕ_1, $\underline{\gamma}$, $\underline{\partial}$, R, D)
20 | **forall the** n' *with* $n \longrightarrow n'$ *and* $\underline{\partial}(n', \Phi) = ??$ **do**
21 | | **if** $(n', \Phi) \in R$ **then**
22 | | | $D := D \cup \{(n', \Phi)\} \cup \{(m, \phi) \in R \mid (m, \phi)$ added to R after $(n', \Phi)\}$
23 | | **else**
24 | | | $b, D :=$ FINDREASON($\underline{\mathcal{A}}$, n', Φ, $\underline{\gamma}$, $\underline{\partial}$, R, D)
25 | | | **if** $b \neq \emptyset$ **then return** b, D
26 **if** $\Phi = \exists \phi_1 \mathsf{U} \phi_2$ **then**
27 | **if** $\underline{\partial}(n, \phi_2) = ??$ **then return** FINDREASON($\underline{\mathcal{A}}$, n, ϕ_2, $\underline{\gamma}$, $\underline{\partial}$, R, D)
28 | **if** $\underline{\partial}(n, \phi_1) = ??$ **then return** FINDREASON($\underline{\mathcal{A}}$, n, ϕ_1, $\underline{\gamma}$, $\underline{\partial}$, R, D)
29 | **forall the** n' *with* $n \longrightarrow n'$ *and* $\underline{\partial}(n', \Phi) = ??$ **do**
30 | | **if** $(n', \Phi) \in R$ **then**
31 | | | $D := D \cup \{(n', \Phi)\} \cup \{(m, \phi) \in R \mid (m, \phi)$ added to R after $(n', \Phi)\}$
32 | | **else**
33 | | | $b, D :=$ FINDREASON($\underline{\mathcal{A}}$, n', Φ, $\underline{\gamma}$, $\underline{\partial}$, R, D)
34 | | | **if** $b \neq \emptyset$ **then return** b, D
35 **return** $\emptyset, D$

but not any other branches of the call tree of FINDREASON. Once we detect a global dependency cycle, we add it to the set D and do no call FINDREASON recursively on that cycle any more but instead try calling it on other successors of n. If no other successors n' for which $\underline{\partial}(n', \Phi) = ??$ exist, the algorithm backtrack to the last nondeterministic choice made, e.g., when choosing a successor node in the $\exists\,\mathsf{U}$ case, by repeatedly returning $\emptyset$, D (see Line 35) until we reach a point where we find another candidate for a recursive call.

This backtracking possibly leads to a box to be contextualized but only if there is a box for which contextualizing may help determining whether $\underline{\mathcal{A}} \models \Phi$ for which the relevant return node is not involved in such a dependency cycle. If no such box if found, we return $\emptyset$, D to GETNEXTEXPANSION. In turn, we try to invoke FINDREASON on the next entry node. If however for all entry nodes FINDREASON returned $\emptyset$, D, we conclude that further progress must be hindered by a global dependency cycle with similar reasoning as in the eager algorithm. For this, we have to slightly adapt the global dependency cycle resolution in LAZYCHECK by changing the following two details:

- the return value of the heuristic is now a pair B, D
- if $B = \emptyset$, the global dependency resolution uses the same logic but may only modify $\underline{\gamma}(ex, \phi)$ if $(ex, \phi) \in D$

We denote the algorithm that implements these two changes as LAZYCHECK$'$. The reason that this modification is necessary is that otherwise we may make $\underline{\gamma}$ inconsistent in case there is an exit node ex and an existential formula ϕ for which $\underline{\gamma}(ex, \phi) = ??$ but that is not on a global dependency cycle but also never contextualized because it is never called

by FindReason as contextualization does not provide any information regarding whether $\underline{\mathcal{A}} \models \Phi$.

Example 13 An example where an infinite chain of calls of FindReason is only prevented by the recursion check via R is in the RSM Fig. 3 when checking against the formula $\Phi = \forall \mathsf{X} \exists \bigcirc \mathsf{U} \bullet$. Here, FindReason would be called with (b_2, n_3) and $\phi = \exists \bigcirc \mathsf{U} \bullet$. In the following steps, FindReason would be invoked with ϕ on n_3, n_6, (b_2, n_6), n_2, and finally the recursion checks in Line 30 via R prevents FindReason being invoked on (b_2, n_6) again. Instead, (b_2, n_3), n_3, n_6, (b_2, n_6) and n_2 along with formula ϕ are all added to D. The backtracking procedure then attempts to find another box but in this case will not succeed since we did not make any non-deterministic choices as ϕ is known in all other branches going from n_1 (i.e., in (n_4) and (b_1, n_1)). Thus, FindReason return $\emptyset$, D. Next, GetNextExpansion would invoke FindReason with another entry node but since n_1 is the only entry node of the initial component, it returns $\emptyset$ as well. As we did not find a box to contextualize with this top-down expansion heuristic, we know we must have a global dependency cycle and thus perform a cycle dependency resolution on D by setting $\underline{\gamma}(n_6, \phi) = \mathbf{ff}$. Note that the global dependency cycle resolution in the lazy algorithm (cf. Line 6 in Algorithm 5) only allows this if the truth value of all subformulas are known in all nodes. For ϕ this is indeed the case, as all subformulas of ϕ are atomic propositions. Further, we are only allowed to modify the value of $\underline{\gamma}(n_6, \phi)$ since $(n_6, \phi) \in D$. Notice that also $\underline{\gamma}(n_5, \phi) = \mathbf{??}$ but $(n_5, \phi) \notin D$, hence we do not modify $\underline{\gamma}(n_5, \phi)$ as we cannot guarantee consistency there. Indeed, notice that despite $\underline{\gamma}(n_5, \phi) = \mathbf{??}$, we can see that n_5 is not even part of a global dependency cycle. In fact, we could easily contextualize b_1 w.r.t.. ϕ to find that $\gamma_{b_1}(n_4, \phi) = \gamma_{b_1}(n_4, \phi) = \mathbf{ff}$ is the only consistent, fully specified context for b_1 w.r.t. ϕ_1 since no node labeled $\bullet$ is reachable from either exit node. However, we can also see that indeed this contextualization is not useful w.r.t. checking whether $\underline{\mathcal{A}} \models \Phi$ as the only way to reach b_1 is through (b_1, n_1) in which we already know that ϕ and Φ hold.

Theorem 3 *Algorithm 5 terminates and is correct for the special expansion heuristic* GetNextExpansion *with the modified global dependency cycle resolution, i.e.,* LazyCheck$'$($\underline{\mathcal{A}}$, Φ, GetNextExpansion) *returns* ***tt*** *iff* $\underline{\mathcal{A}} \models \Phi$ *and* ***ff*** *iff* $\underline{\mathcal{A}} \not\models \Phi$ *for any RSM* $\underline{\mathcal{A}}$ *and CTL formula* Φ *over a common set of atomic propositions.*

6.4 Problem instances

Having introduced the generalized lazy model-checking scheme for RSMs, we now investigate the impact of our algorithm on special instances of the CTL model-checking problem for RSMs and compare them to the eager approach from a theoretical point of view.

6.4.1 Guaranteed exponential succinctness of the lazy approach

It is well known that the CTL model-checking problem for RSMs is ExpTime-complete [3], which directly provides exponential worst-case time complexity of our algorithms. However, we can construct a CTL formula Φ and a class of RSMs $\mathfrak{A}$ for which the lazy model checking algorithm requires constant contextualization steps, regardless of the expansion heuristic, while the eager approach requires exponentially many contextualizations for every RSM in $\mathfrak{A}$. This directly implies polynomial run time of the lazy algorithm in the size of the RSM but exponential run time for the eager approach on this specific class of problem instances.

To construct such an example, take an RSM $\underline{\mathcal{A}}$ and a CTL formula Φ for which the eager approach requires building exponentially many contexts with respect to the size of Φ. Such instances must exist, since the CTL model-checking problem for RSMs is ExpTime-complete while the local CTL deduction on finite Kripke structures is doable in polynomial time. We then construct a modified RSM $\underline{\mathcal{A}}'$ by assigning a fresh label $\bullet$ to all initial nodes. When checking whether $\underline{\mathcal{A}}' \models \Phi \vee \bullet$, the eager approach still requires exponential time w.r.t the size of the RSM since it performs a bottom-up search and Φ is a direct subformula of $\Phi \vee \bullet$. Meanwhile, the lazy approach (with any heuristic) is guaranteed to determine that $\underline{\mathcal{A}}' \models \Phi \vee \bullet$ in linear time in the size of the RSM without any contextualization due to the ternary model checking that operates top-down and terminates directly after evaluating $\bullet$ locally.

While this of course does not hold for all problem instances due to the ExpTime-completeness of the model-checking problem, it shows that there is a non-empty subclasses of the model-checking problem for which the lazy algorithm asymptotically outperforms the eager algorithm.

6.4.2 Bounding exit and entry nodes

Single-exit and *single-entry RSMs* constitute the most prominent subclasses of RSMs that have been considered toward efficient model-checking algorithms. In general, we say that an RSM $\mathcal{A}$ is *k-entry* or *k-exit* depending on a parameter $k \in \mathbb{N}$ if for each component $\mathcal{A}_i \in \mathcal{A}$ we have $|En_i| \leq k$ or $|Ex_i| \leq k$, respectively. It is known that for 1-exit RSMs the CTL model checking with a fixed formula is solvable in linear time in the size of the RSM [3]. The main insight we can draw from this restriction to a single-exit node n per component is that at most three distinct contexts have to be constructed for each component as they can only map n to either **tt**, **ff**, or **??**. This reasoning can easily be extended to RSMs with a

fixed parameter k that bounds the number of exit nodes per component and hence also puts a constant (albeit exponential in k) bound on the number of different contexts. Thus, for a fixed parameter k, the CTL model-checking problem on k-exit RSMs has linear-time parametrized complexity. Clearly, the parametrized complexity agrees for both, lazy and eager RSM model checking, since the global deduction necessary for both approaches runs in linear time with respect to the size of the RSM. It is known that restrictions on the number of entry nodes of a component do not influence the complexity of the model-checking problem [3], hence bounding or parametrizing them does not have any impact on the asymptotic complexity of our algorithms neither.

6.4.3 Single-component and single-box RSMs

A natural question concerning subclasses of RSMs is whether there is a class of RSMs restricted only on the parameters of the RSM, like for k-exit RSMs, but not its structure, for which the lazy model-checking scheme requires strictly less contextualization steps than the eager algorithm.

One might consider RSMs consisting of a single component; however, every RSM can be transformed into an equivalent single-component RSM. Moreover, its size is linear in the size of the original RSM. This follows from the transformations given in [11] with which we can linearly transform any RSM into a PDS and the resulting PDS back to an RSM which by way of the transformation consists of only a single component.

A stronger restriction we may impose is to only allow for a single box b. Notice that this directly implies that there is only one component (or there is an equivalent RSM with only one component), since the box must be in the initial component in order to be reachable. If $Y(b) = 1$ then no other component is reachable and as such they may be removed. If $Y(b) = i > 1$ then the box may be replaced by $\mathcal{A}_i$, resulting in an equivalent box-less RSM.

Box stacks of single-box RSMs, where b is the box in the RSM, obviously only consist of strings of the form b^* and can thus be uniquely characterized by the size of the box stack. Thus, they correspond to *one-counter automata* (OCA) [31], which are a variation on PDSs where instead of a tape we have a counter. This intuitive correspondence by can be shown by transformations between single-box RSMs and OCA. A linear transformation from OCA to single-box RSM follows immediately from the general PDS to RSM transformation given in [11] since an OCA can equivalently be expressed as a PDS with a single tape symbol.

We show that there is a linear transformation from OCA to single-box RSM as well, which can be obtained by modifying the transformation in [11] to encode the current node in the state of the PDS, rather than in the tape, i.e., for a single-box RSM $\underline{\mathcal{A}} = \mathcal{A}_1 = (N, \{b\}, Y, \{en_1, \ldots, en_l\}, \{ex_1, \ldots, ex_k\}, \longrightarrow, AP, L)$ we define an equivalently labeled PDS $\mathcal{P} = (Q, \Gamma, \longrightarrow', AP, L)$ with

- $Q = N \cup (\{b\} \times En) \cup (\{b\} \times Ex)$
- $\Gamma = \{b\}$
- $\longrightarrow'$ such that
  - $n_1 \longrightarrow' n_2$ if $n_1 \longrightarrow n_2$
  - $(b, en_i) \longrightarrow' n\ b$ if $en_i \longrightarrow n$
  - $ex_i\ b \longrightarrow' n$ if $(b, ex_i) \longrightarrow n$
  - all transitions of $\longrightarrow'$ are defined by the above rules

This now gives us a linear transformation from single-box RSM to a PDS with a single tape symbol, and thus to an OCA. Notice that this transformation is indeed linear for single-box RSM as otherwise for arbitrary RSMs the state space of the PDS would be quadratic in the size of the RSM.

Since the CTL model-checking problem for OCA is PSPACE-complete for a fixed formula [31, Theorem 7.2], as well as for a fixed system [31, Theorem 5.8], our transformation is sufficient to show via a polynomial (even linear) reduction that the same results hold for single-box RSM.

Hence, unless P = PSPACE, the CTL model-checking problem for a fixed formula over single-box RSMs is not solvable in polynomial time. In particular, this means that the lazy approach most likely does not perform asymptotically better than the eager approach in this subclass of RSMs.

6.4.4 Summary

To sum up, while the general problem of model checking RSMs against CTL formulas is EXPTIME-complete, we have seen that restricting the RSM to a constant number of exit nodes per component renders the problem to have parametrized polynomial time complexity. Additionally, for single-box RSMs, the problem is PSPACE-complete.

In general, while it is possible to specifically construct classes of problem instances in which the lazy approach outperforms the eager approach by an exponential factor, we do not expect that there is a "natural" restriction on RSMs or CTL formulas such that model-checking problem with the lazy evaluation scheme performs asymptotically better than with the eager algorithm. Nonetheless, in the next section we will show that the lazy approach yields a significant speed-up in practice in various experimental studies.

7 Implementation and evaluation

We implemented both the eager and the lazy approach presented in this paper in a prototypical tool RSMCHECK. Written in PYTHON3, it is supported by almost all com-

mon 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 [33] or PUMOC [49] that follows the standard translation method (see, e.g., [11]).

7.1 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:

1. Is our lazy approach effective, i.e., generates significantly less contexts and is faster compared to the eager approach?
2. How do analysis times of our approaches implemented in RSMCHECK compare to state-of-the-art procedural model checkers?
3. Can real-world procedural programs be verified with our approaches?

Specifically, here we called EAGERCHECK the eager approach while the lazy approach refers to LAZYCHECK with GETNEXTEXPANSION and as expansion heuristics with correct global cycle resolution as discussed at the end of Sect. 6.3.

7.1.1 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 min and a memory limit of 4 GB of RAM. Compared to the experimental studies we conducted in the conference version [25] of this article, we run the experiments on an updated version of RSMCHECK where for the eager approach, optimized data structures using hashing were used. This explains the potentially arising speedups for EAGERCHECK of up to 40%.

7.2 Scalability experiment

First, we conducted a scalability experiment to compare the eager 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%, and the formula Φ_j has a quantifier depth of $\lfloor j/9 \rfloor$. Note that the chosen degree of connectivity is sufficient to obtain indirect recursions in the majority of cases. Figure 8 shows the analysis times in seconds for our lazy (top) and eager (bottom) approach. We observe that the more compositional structure and the bigger the requirement formulas, the more the lazy approach pays off compared to the eager approach, both in memory consumption and analysis speed. In 5% of the cases, the eager approach ran into memouts and in all other cases the lazy approach is on average eight times faster than the eager one. For 1 we conclude that lazy contextualization is an effective method that allows for faster RSM model checking.

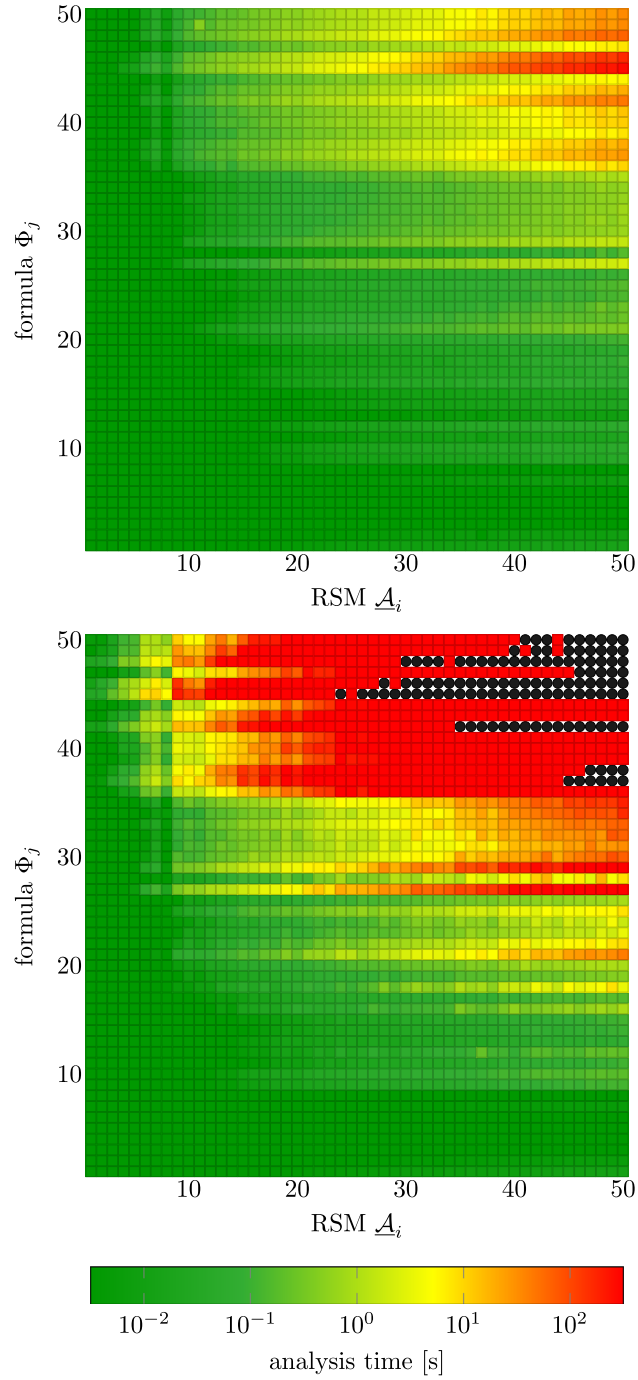


Fig. 8 Analysis times for the scalability experiment in seconds (logarithmic scale, lazy on the top, eager on the bottom, • stands for memout)

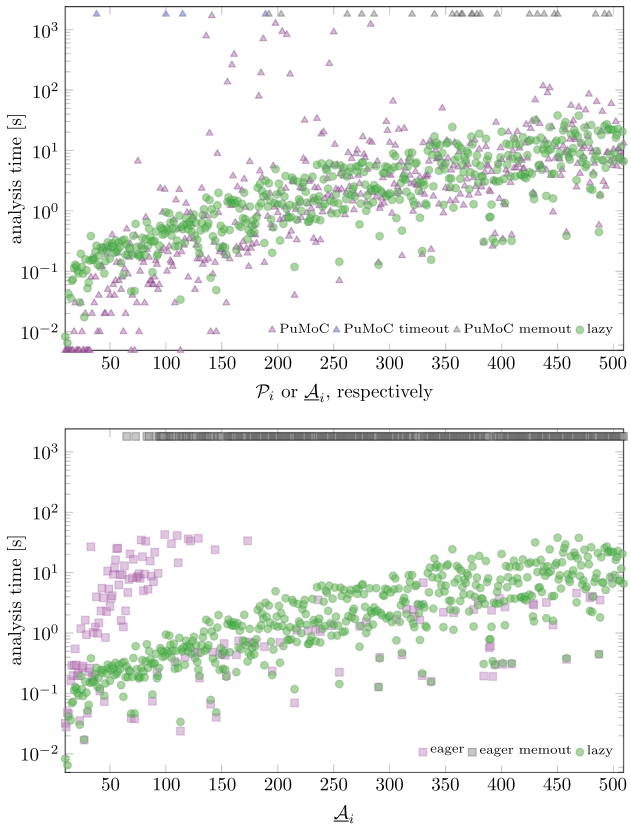


Fig. 9 Analysis times for 500 PuMoC examples in seconds (logarithmic scale)

7.3 PuMoC benchmark set

Our second experimental study compares RSMCheck to the procedural CTL model checker PuMoC on its benchmark set [49], comprising 500 randomly generated pushdown systems. The pushdown systems $\mathcal{P}_i$ and CTL formulas Φ_i are numbered as in [49] with $i \in \{10, 11, \ldots, 509\}$ where sizes 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, as our results in Fig. 9 show, while PuMoC runs into time- or memouts in 28 examples, the lazy approach successfully completes each experiment in less than 40 s. Most of the analysis times are in the same range (see Fig. 9 on the top) even though PuMoC is implemented in C, while RSMCheck is implemented in Python, known for broad applicability but comparably weak performance. Regarding 2, we can conclude that RSMCheck is competitive with the state-of-the-art model checker PuMoC even on single-component RSMs. Figure 9 on the bottom shows a comparison of the lazy approach to the eager one, applied on the 500 PuMoC examples. The eager approach is almost always equally fast or slower compared to the lazy approach and runs into memouts in 69% of the cases.

All the few cases in which the eager approach outperforms the lazy approach are on formulas in which no quantified path formulas occur, i.e., which can locally be solved in the entry node. As model checking on these formulas does not require any contextualization, the lazy and eager approach are equivalent and both solve the problem in the order of 0.1 seconds or less. We do not see that either approach is consistently faster in those cases, and the difference is always negligible (0.03 s or less) so we attribute these results to small fluctuations in performance, or time measurement. Importantly, in all "interesting" cases, i.e., where the CTL formula contains at least one quantifier, the lazy approach outperforms the eager approach.

This also supports our positive answer to 1 drawn in the last section.

7.4 Interprocedural static analysis for Java programs

Our last experimental study considers an interprocedural analysis for real-world systems, borrowed from the benchmark set of [33], containing examples from the AVR Simulation and Analysis Framework (avrora) [2], as well as the Apache™ FOP Project (dom2pdf and fo2pdf) [1]. 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. To give an estimate of the size of the model, column k of Table 1 indicates the number of reachable components of the RSM for the Java program. In all examples, with the exception of the Dataflow toy example (Fig. 1), we observe recursive function calls, resulting in non-trivial cyclic paths through boxes in the RSM.

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 we check formalizes that whenever the selected variable is defined, it is eventually used (see the Dataflow example in the preliminaries).

Additionally to the lazy evaluation scheme, with GetNextExpansion as expansion heuristic, we also consider the *ternary evaluation scheme* by which we refer to the expansion heuristic $H(\underline{\mathcal{A}}, \Phi, \underline{\partial}) = C(\underline{\mathcal{A}}, \phi)$. This heuristic is of particular interest, since it encapsulates the eager expansion strategy of EagerCheck while still avoiding potentially unnecessary contextualization steps by employing ternary model checking.

Table 1 shows characteristics of our analysis. First, the lazy, ternary, and even the eager approach are significantly faster than PDSolver and PuMoC. Firstly, as the examples are real-world models we can positively answer 3.

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

JAVA program	Result	PDSOLVER time	PUMOC time	k	Eager		Ternary		Lazy	
					#ctx	Time	#ctx	Time	#ctx	Time
Dataflow (Fig. 1)	**ff**	< 0.01	0.02	3	6	< 0.01	1	< 0.01	1	< 0.01
avroraCFG	**tt**	> 1800	> 1800	3169	4372	107.53	160	56.73	2	8.22
avroraDisassemble	**tt**	806.66	> 1800	2085	4628	163.89	1731	118.92	1	3.47
avroraELF	**tt**	26.68	71.28	248	614	4.79	1	0.29	1	0.29
avroraMedTest	**tt**	12.48	37.09	238	264	1.32	28	0.69	4	0.43
avroraReg	**tt**	8.73	16.12	173	477	1.77	27	0.56	2	0.32
dom2pdf	**tt**	80.46	1345.56	615	2002	17.64	240	6.93	1	0.79
fo2pdf	**tt**	61.68	> 1800	607	2029	27.67	60	1.42	6	1.23

Further, contributing to 2, 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 eager approach generates only those contexts that arise during deduction steps in exit nodes. These studies also support that our lazy approach is effective (cf. 1): Column #ctx indicates the number of generated contexts during analysis, which is significantly lower than the number of components in the RSM (cf. column k). Hence, we can observe that the lazy approach effectively avoids context generation for many of the components, having a direct impact on the analyzed state spaces and timings. This effect is most emphasized with the GETNEXTEXPANSION heuristic. Further, in the lazy approach we observe speedups of up to two orders of magnitude compared to the eager approach.

For the performance of the ternary approach, we observe two cases: For examples where the initial contextualization is already enough do determine whether $\underline{\mathcal{A}} \models \Phi$ (i.e., Dataflow and avroraELF), the ternary evaluation scheme effectively avoids any further contextualization steps, thus making the ternary approach equivalent to the lazy one. In the other examples we notice that the ternary approach generally builds less contexts and takes less time than the eager algorithm but is outperformed by the lazy approach. One reason for this that we could observe when looking at the experiments in detail (besides the lazy scheme clearly building less contexts overall) was that the lazy approach being able to initiate a global cycle resolution step earlier than the ternary approach. This is due to the ternary approach requiring all subformulas to be known in every node before performing a global cycle resolution while the lazy approach utilizing the FINDREASON as a guided heuristic only requires subformulas to be known in nodes which actually may influence the satisfaction relation. This is most emphasized in the examples avroraDisassemble and dom2pdf, where the lazy scheme can decide the model checking problem only by global cycle resolution rather than contextualization, whereas the ternary scheme requires several contextualization steps before resolving global cycles.

7.4.1 Random expansion heuristic

To show the impact of using the guided heuristic GETNEXTEXPANSION, we additionally ran all examples in Table 1 with an expansion heuristic that in each step picks a random box $b \in C(\underline{\mathcal{A}}, \underline{\partial})$ to contextualize. Similar to the ternary expansion heuristic, we observe that the approach is equivalent to lazy checking if the model checking problem can be solved locally in each component, i.e., without any contextualizations. For examples where a global cycle check is required, we observe the same situation as in the ternary case where all boxes need to be contextualized before global cycles can be resolved. The major difference is that the ternary approach contextualizes all boxes at once before calling REFINETERNARY, while the random heuristics calls REFINETERNARY after each contextualization. As the global deduction is the most expensive solution, the runtime in avroraDisassemble was over 3.5 h in all of our runs. For the examples where no global cycle resolution is necessary, we naturally observed fluctuating runtimes, depending on the boxes chosen at random. Still in those examples, the random heuristic is outperformed in every example by the ternary and lazy approach by up to an order of magnitude.

Overall, we conclude that the guided heuristic as well as the improved global cycle resolution of GETNEXTEXPANSION improve the runtime immensely and that the overhead due to GETNEXTEXPANSION is negligible.

8 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 com-

pared 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. [3] for verifying CTL formulas.

8.1 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 FINDREASON 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.

8.1.1 Heuristics for non-deterministic choices

Central in our lazy approach is the nondeterministic algorithm FINDREASON, which determines the next context to be considered. This algorithm leaves some freedom in how the non-determinism 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 3 of Algorithm 6. 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 to resolve the non-determinism did not significantly change runtimes, which is explainable since the CTL requirements were either randomly generated or a comparably simple use-def formula. More complex settings could also benefit from integrating advanced techniques to explain the reasons for unknown formula evaluations [7].

8.1.2 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., [3, 6]). 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., [45]) equip RSMs with labels from a semiring, similarly as *probabilistic RSMs* equip transitions with probabilities (see, e.g., [11, 27]).

Another direction would be to investigate the influence of the ternary *thorough semantics* [13] on our RSM algorithms, where unknown state interpretations require at least one candidate for each, fulfilling and not fulfilling state satisfaction. However, the latter approach might lead to a higher computational complexity, an impact to be evaluated along with the alternative semantics.

SAT-based model checking [8] may outperform classical model-checking techniques also in the lazy abstraction setting with interpolation [42, 51]. An interesting avenue would be to include the compositional structure and our lazy evaluation schemes into a SAT-based model checker, with potential benefits of state-of-the-art advances in satisfiability checking.

Acknowledgements The authors were partly 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). The first author has been further supported by NWO Veni grant VI.Veni.222.431. Thanks go also to the anonymous reviewers who helped to improve the quality of the manuscript.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

Appendix

In this appendix, we provide the proofs of the main paper.

A Ternary RSM model checking

A.1 Ternary refinement

Lemma 1 *Let $\underline{\mathcal{A}}$ be an RSM with a consistent context $\underline{\gamma}$ over a CTL formula Φ. Then the ternary interpretation induced by the ternary semantics given by*

$$\underline{\partial}(n, \phi) \;=\; [n \models \phi]_{\underline{\gamma}}$$

for all $n \in \mathsf{N}^{all}$ and $\phi \in Subf(\Phi)$ is consistent.

Proof We show this by induction on the CTL formula Φ. In the following, let $n \in \mathsf{N}^{all}$ be an arbitrary node.

For the base case, $\Phi = a$ where $a \in AP$ the statement clearly holds.

If $\Phi = \neg\phi_1$ and $[n \models \phi_1]_{\underline{\gamma}} = \mathbf{tt}$ we have that $[n \models \Phi]_{\underline{\gamma}} = \mathbf{ff}$. By IH we have that $\underline{\partial}$ is consistent for formula ϕ_1 and thus it holds that $(\sigma, n) \models \phi_1$, hence $(\sigma, n) \not\models \Phi$ for all $\sigma \in B^*$, which renders $\underline{\partial}$ to be consistent. Analogously we obtain consistency of $\underline{\partial}$ for $[n \models \phi_1]_{\underline{\gamma}} = \mathbf{ff}$. If $[n \models \phi_1]_{\underline{\gamma}} = \mathbf{??}$ then $[n \models \Phi]_{\underline{\gamma}} = \mathbf{??}$ and thus consistency of $\underline{\partial}$ is trivial, as both implications required to hold for consistency of $\underline{\partial}$ trivially hold as none of the preconditions are satisfied.

Now consider the case $\Phi = \phi_1 \vee \phi_2$. If $[n \models \phi_1]_{\underline{\gamma}} = \mathbf{tt}$ or $[n \models \phi_2]_{\underline{\gamma}} = \mathbf{tt}$ then $[n \models \Phi]_{\underline{\gamma}} = \mathbf{tt}$. W.l.o.g. assume $[n \models \phi_1]_{\underline{\gamma}} = \mathbf{tt}$. Then by IH $(\sigma, n) \models \phi_1$ and hence $(\sigma, n) \models \Phi$ for all $\sigma \in B^*$. Thus, $\underline{\partial}$ is consistent in this case. A similar argument holds when $[n \models \phi_1]_{\underline{\gamma}} = \mathbf{ff}$ and $[n \models \phi_2]_{\underline{\gamma}} = \mathbf{ff}$. In the remaining cases where one of $[n \models \phi_1]_{\underline{\gamma}}$ and $[n \models \phi_2]_{\underline{\gamma}}$ is **??** and the other is either **ff** or **??** we have $[n \models \Phi]_{\underline{\gamma}} = \mathbf{??}$ and thus consistency of $\underline{\partial}$ is again trivial.

For existentially quantified formulas, first consider case $\Phi = \exists\mathsf{X}\phi_1$. If n is an exit node, $[n \models \Phi]_{\underline{\gamma}} = \underline{\gamma}(n, \Phi)$ and consistency trivially follows from the assumption that $\underline{\gamma}$ in consistent. If n is not an exit node, we follow a similar reasoning as in the disjunctive case. If $[n', \phi_1]_{\underline{\gamma}} = \mathbf{tt}$ for some $n \longrightarrow_+ n'$ then $[n \models \Phi]_{\underline{\gamma}} = \mathbf{tt}$ By IH $(\sigma, n') \models \phi_1$ and thus $(\sigma, n) \models \Phi$ for all $\sigma \in B^*$. Hence $\underline{\partial}$ is consistent. Analogously again, if $[n', \phi_1]_{\underline{\gamma}} = \mathbf{ff}$ for all $n \longrightarrow_+ n'$ then $[n \models \Phi]_{\underline{\gamma}} = \mathbf{ff}$ and again by IH and definition of $\exists\mathsf{X}$ we have $(\sigma, n) \not\models \Phi$ for all $\sigma \in B^*$ implying $\underline{\partial}$ is consistent. For the remaining cases we have $[n \models \Phi]_{\underline{\gamma}} = \mathbf{??}$, which again is trivially consistent.

Now consider $\Phi = \exists\mathsf{G}\phi_1$. Here, we can have two ways for $[n_0, \Phi]_{\underline{\gamma}} = \mathbf{tt}$: The first possibility is that there is a cycle $n_0 \longrightarrow_+ n_1 \longrightarrow_+ n_2 \longrightarrow_+ \ldots \longrightarrow_+ n_l \longrightarrow_+ n_0$ for which $[n_i, \phi_1]_{\underline{\gamma}} = \mathbf{tt}$ for all $i \in \{0, 1, \ldots, l\}$. Then by IH for each node n_i on the cycle we have $(\sigma, n_i) \models \phi_1$ and thus $(\sigma, n) \models \Phi$ for all $\sigma \in B^*$. Hence, $\underline{\partial}$ is consistent. The second possible reason for $[n_0, \Phi]_{\underline{\gamma}} = \mathbf{tt}$ is if there is a path $n_0 \longrightarrow_+ n_1 \longrightarrow_+ n_2 \longrightarrow_+ \ldots \longrightarrow_+ n_l$ for which $[n_i, \phi_1]_{\underline{\gamma}} = \mathbf{tt}$ for all $i \in \{0, 1, \ldots, l\}$ and $n_l \in Ex$ with $\underline{\gamma}(n_l, \Phi) = \mathbf{tt}$. Then by IH for each node n_i on the path we have $(\sigma, n_i) \models \phi_1$ for all $\sigma \in B^*$. Additionally, by consistency of $\underline{\gamma}$ and $\underline{\gamma}(n_l, \Phi) = \mathbf{tt}$ we have a $(\sigma, n_l) \models \Phi$. Be definition of $\exists\mathsf{G}$ we thus have $(\sigma, n) \models \Phi$ for all $\sigma \in B^*$ and hence consistency of $\underline{\partial}$. We again give a similar argument in case $[n_0, \Phi]_{\underline{\gamma}} = \mathbf{ff}$. This can only happen if for every cycle $n_0 \longrightarrow_+ n_1 \longrightarrow_+ n_2 \longrightarrow_+ \ldots \longrightarrow_+ n_l \longrightarrow_+ n_0$ there is an n_i for which $[n_i, \phi_1]_{\underline{\gamma}} = \mathbf{ff}$ and additionally for every path $n_0 \longrightarrow_+ n_1 \longrightarrow_+ n_2 \longrightarrow_+ \ldots \longrightarrow_+ n_l$ with $n_l \in Ex$ there is an n_i for which $[n_i, \phi_1]_{\underline{\gamma}} = \mathbf{ff}$, or $\underline{\gamma}(n_l, \Phi) = \mathbf{ff}$. In either case, by definition of $\exists\mathsf{G}$ we obtain that $(\sigma, n_i) \not\models \Phi$ for all $\sigma \in B^*$ and thus $\underline{\partial}$ is consistent. Lastly, if $[n \models \Phi]_{\underline{\gamma}} = \mathbf{??}$ consistency of $\underline{\partial}$ is trivial.

The argumentation for $\Phi = \exists\phi_1 \mathsf{U} \phi_2$ is analogous to the case $\Phi = \exists\mathsf{G}\phi_1$ except that we do not consider cycles, but finite paths that lead up to a state in which ϕ_2 holds. □

Lemma 2 *Let $\underline{\mathcal{A}}$ be an RSM with context $\underline{\gamma}$ over a CTL formula Φ such that $\underline{\gamma}(n, \phi) \neq$ **??** for all $n \in Ex$ and $\phi \in Subf_{\exists}(\Phi)$. Then the ternary interpretation induced by the ternary semantics given by*

$$\underline{\partial}(n, \phi) = [n \models \phi]_{\underline{\gamma}}$$

for all $n \in \mathsf{N}^{all}$ and $\phi \in Subf(\Phi)$ is maximally consistent.

Proof We first show $[n \models \phi]_{\underline{\gamma}} \neq \mathbf{??}$ for all $n \in \mathsf{N}^{all}$ and $\phi \in Subf_{\exists}(\Phi)$ by induction on Φ.

In the base case, if $\Phi = a$ for some $a \in AP$, the statement trivially holds.

For Φ being a negation or disjunction, this directly follows from the IH and the fact that $\{\mathbf{tt}, \mathbf{ff}\}$ is closed under *comp* and max.

If Φ is an existential formula, $[n \models \phi]_{\underline{\gamma}} \neq \mathbf{??}$ is a combination of minima and maxima of terms of form $[n \models \phi]_{\underline{\gamma}}]$ with $\phi \in Subf_{\exists}(\Phi) \setminus \{\Phi\}$, and terms of form $\underline{\gamma}(n, \Phi)$ which are all **tt** or **ff** due to IH or assumption, respectively. Hence, any combination of those using only min and max is also **tt** or **ff**, and in particular not **??**, showing the claim.

Since $\underline{\partial}$ is consistent due to Lemma 1, it only remains to show that $\underline{\partial}$ is maximally consistent. For this, fix a node $n \in \mathsf{N}^{all}$. Assume that $(n, \sigma) \models \Phi$ for all box stacks $\sigma \in B^*$. Then $\underline{\partial}(n, \phi)$ cannot be **ff** as consistency of $\underline{\partial}$ would imply that $(n, \sigma) \models \Phi$ for all box stacks $\sigma \in B^*$. Also, $\underline{\partial}(n, \phi) = [n \models \Phi]$, which is not **??** and hence $\underline{\partial}(n, \phi) = \mathbf{tt}$.

Analogously, we have that from $(n, \sigma) \not\models \Phi$ for all box stacks $\sigma \in B^*$ it follows that $\underline{\partial}(n, \phi) = \mathbf{tt}$. By definition, this leads to $\underline{\partial}$ being maximally consistent. □

Lemma 3 *Let $\underline{\mathcal{A}} = (\mathcal{A}_1, \ldots, \mathcal{A}_k)$ be an RSM $\underline{\mathcal{A}}$ with consistent contexts $\underline{\gamma}$ over a CTL formula Φ. Further, let $\underline{\partial}$ be the ternary interpretation induce by the ternary semantics for all $n \in \mathsf{N}^{all}$ and $\phi \in Subf(\Phi)\backslash\{\Phi\}$, i.e., $\underline{\partial}(n, \phi) = [n \models \phi]_{\underline{\gamma}}$*

Then Algorithm 1 terminates and returns a ternary interpretation $\underline{\partial}'$ that follows the ternary semantics for all $n \in \mathsf{N}^{all}$ and $\phi \in Subf(\Phi)$, i.e., $\underline{\partial}'(n, \phi) = [n \models \phi]_{\underline{\gamma}}$.

Proof This lemma is shown mainly by an inductive case distinction on ternary semantics definitions. Let us first consider the case where $n \in \mathsf{N}^{all}$ and $\phi \in Subf(\Phi)\backslash\{\Phi\}$. As we set $\underline{\partial}'(n, \phi) = \underline{\partial}(n, \phi)$ in Line 2, and because $\underline{\partial}'(n, \phi)$ is never modified after, from our assumption it follows that $ctxi'(n, \phi) = \underline{\partial}(n, \phi) = [n \models \phi]_{\underline{\gamma}}$ and thus the statement holds.

Knowing that the statement holds for all $\phi \in Subf(\Phi) \setminus \{\Phi\}$, we now show that the statement also holds for Φ itself.

Case I: Φ is propositional or $\Phi = \exists \mathsf{X}\phi_1$.

If Φ is a propositional formula, i.e, an atomic proposition, a negation or a disjunction, we can immediately see that REFINETERNARY follows the ternary semantics by definition. Similarly, if $\Phi = \exists \mathsf{X}\phi_1$ we also simply compute $\underline{\partial}'(n, \phi)$ by definition by either copying $\underline{\gamma}(n, \Phi)$ if n is an exit node, or else taking the maximum of $\underline{\partial}(n', \phi_1) = [n', \phi_1]_{\underline{\gamma}}$ over all $\longrightarrow_+$-successors n'.

Case II: $\Phi = \exists \mathsf{G}\phi_1$.

Case II.1: $n \in Ex$.

Assume $\underline{\gamma}(n, \Phi) = \mathbf{tt}$. Then $n \in Sat_{pes}$ and $n \in Sat_{opt}$ after the initialization in Line 15 and Line 15, respectively. In the following while loop, only $n \notin Ex$ are considered and thus we still have $n \in Sat_{pes}$ and $n \in Sat_{opt}$ and hence $\underline{\partial}'(n, \Phi) = \mathbf{tt}$. Analogously, if $\underline{\gamma}(n, \Phi) = \mathbf{ff}$ then initially $n \notin Sat_{pes}$ and $n \notin Sat_{opt}$ and they are never added as we never add any elements to either Sat_{opt} or Sat_{pes} in the $\exists \mathsf{G}$-branch and thus $\underline{\partial}'(n, \Phi) = \mathbf{ff}$. Similarly, if $\underline{\gamma}(n, \Phi) = \mathbf{??}$ then $n \notin Sat_{pes}$ but $n \in Sat_{opt}$ and hence $\underline{\partial}'(n, \Phi) = \mathbf{??}$. We can summarize these three statements as $\underline{\partial}'(n, \Phi) = \underline{\gamma}(n, \Phi)$. As by definition of RSMs, exit node do not have any successors, there are no n' where $n \longrightarrow_+ n'$ and thus the only path starting in n is the trivial path of length 1 consisting of only n itself. In this case the ternary semantics simplifies to $[\cdot, \Phi]_{\underline{\gamma}} = \underline{\gamma}(n, \Phi)$, matching $\underline{\partial}'$.

Case II.2: $n \notin Ex$.

Case II.2.1: $[n \models \Phi]_{\underline{\gamma}} = \mathbf{tt}$.

This implies that either there is a path $n \longrightarrow_+ n_1 \longrightarrow_+ \ldots \longrightarrow_+ n_i \longrightarrow_+ \ldots \longrightarrow_+ n_i$ such that $[\cdot, \phi_1]_{\underline{\gamma}} = \mathbf{tt}$ for all nodes on the path, or a path $n \longrightarrow_+ n'_1 \longrightarrow_+ \ldots \longrightarrow_+ ex$ such that $ex \in Ex$ with $\underline{\gamma}(ex, \Phi) = \mathbf{tt}$ and $[\cdot, \phi]_{\underline{\gamma}} = \mathbf{tt}$ for all other nodes on the path. In REFINETERNARY, initially all nodes on this path are included in Sat_{pes} and Sat_{opt}. As witnesses by this path, each non-exit node on the path has a $\longrightarrow_+$-successor in Sat_{pes} and Sat_{opt} and is thus never removed. Additionally, exit nodes are never removed from Sat_{pes} and Sat_{opt} anyway. Hence we set $\underline{\partial}'(n, \Phi) = \mathbf{tt} = [n \models \Phi]_{\underline{\gamma}}$.

Case II.2.2: $[n \models \Phi]_{\underline{\gamma}} = \mathbf{ff}$.

To be in this case, we must have that all paths $n \longrightarrow_+ n_1 \longrightarrow_+ \ldots \longrightarrow_+ n_i \longrightarrow_+ \ldots \longrightarrow_+ n_i$ contain at least one node n_j with $[n_j, \phi_1]_{\underline{\gamma}} = \mathbf{ff}$ and for all paths $n \longrightarrow_+ n'_1 \longrightarrow_+ \ldots \longrightarrow_+ ex$ with $ex \in Ex$ we either have $\underline{\gamma}(ex, \Phi) = \mathbf{ff}$ or some other node n'_j on the path with $[n'_j, \phi_1]_{\underline{\gamma}} = \mathbf{ff}$.

If $[n \models \phi_1]_{\underline{\gamma}} = \mathbf{ff}$, then n is never added to Sat_{opt} and trivially $\underline{\partial}'(n, \Phi) = \mathbf{ff}$. Otherwise, if $\underline{\partial}'(n, \Phi) \neq \mathbf{ff}$, then n is still in Sat_{opt} initially. However, we now show that it is removed from Sat_{opt} after a finite number of iterations of the while-loop in the optimistic run. As there is no cycle on which $[\cdot, \phi_1]_{\underline{\gamma}} \neq \mathbf{ff}$ and no path to an exit node ex for which $\underline{\gamma}(ex, \Phi) \neq \mathbf{ff}$ and $[\cdot, \phi_1]_{\underline{\gamma}} \neq \mathbf{ff}$ for all other nodes on the path, there is a finite path $n \longrightarrow_+ n_1 \longrightarrow_+ \ldots \longrightarrow_+ n_l$ that is maximal in its length for which $[\cdot, \phi_1]_{\underline{\gamma}} \neq \mathbf{ff}$ on all nodes on the path and that does not contain any exit nodes.

We inductively show that after m iterations of the while-loop in the optimistic run, the longest $\longrightarrow_+$-path in Sat_{opt} starting in n has length $l + 1 - m$. For $m = 0$ this trivially holds as the maximal path on which $[\cdot, \phi_1]_{\underline{\gamma}} = \mathbf{ff}$ holds has length $l + 1$ and only nodes for which $[n_i, \phi_1]_{\underline{\gamma}} = \mathbf{ff}$ are initially added to Sat_{opt}. After m iterations, let $n \longrightarrow_+ n_1 \longrightarrow_+ \ldots \longrightarrow_+ n_{l-m}$ be a longest $\longrightarrow_+$-path in Sat_{opt}. We show that after $m+1$ iterations, be longest $\longrightarrow_+$-path in Sat_{opt} has length $l-m$. Notice that n_{l-m} cannot have any non-exit node successor in Sat_{opt} as otherwise we could have found a longer path. Further, n_{l-m} can also not have an exit node ex as successor in Sat_{opt} as the inclusion of ex in Sat_{opt} would imply $\underline{\gamma}(ex, \Phi) \neq \mathbf{ff}$. But then $n \longrightarrow_+ n_1 \longrightarrow_+ \ldots \longrightarrow_+ n_{l-m} ex$ would be a path from n to an exit node ex with $\underline{\gamma}(ex, \Phi) \neq \mathbf{ff}$ and $[\cdot, \phi_1]_{\underline{\gamma}} \neq \mathbf{ff}$ for all other nodes on the path, which we assumed does not exist. This means n_{l-m} has no successors in Sat_{opt} and is thus removed in the $m + 1$-th iteration of the while loop. This argument holds for all paths of length $l - m$, should there be multiple. Thus, after the $m + 1$-th iteration of the while loop, the longest path in Sat_{opt} starting in n has length $l - m$, finishing the proof.

This means that after l iterations, the longest $\longrightarrow_+$-path in Sat_{opt} starting in n has length 1, i.e., is just n itself. This means n has no $\longrightarrow_+$-successors in Sat_{opt} and thus is removed from Sat_{opt} in the $l + 1$-th iteration. Hence, $\underline{\partial}'(n, \Phi) = \mathbf{ff}$.

Case II.2.3: $[n \models \Phi]_{\underline{\gamma}} = \mathbf{??}$.

For this, two assumptions must hold:

(1) all paths starting in n reaching a cycle contain at least one node n_j with $[n_j, \phi_1]_{\underline{\gamma}} \neq \mathbf{tt}$ and all paths from n to an arbitrary exit node $ex \in Ex$ either have $\underline{\gamma}(ex, \Phi) \neq \mathbf{tt}$ or some other node n_j on the path with $[n_i, \phi_1]_{\underline{\gamma}} \neq \mathbf{tt}$, but

(2) there is a path $n \longrightarrow_+ n_1 \longrightarrow_+ \ldots \longrightarrow_+ n_i \longrightarrow_+ \ldots \longrightarrow_+ n_i$ such that $[\cdot, \phi_1]_{\underline{\gamma}} \neq \mathbf{ff}$ for all nodes on the path, or a path $n \longrightarrow_+ n'_1 \longrightarrow_+ \ldots \longrightarrow_+ ex$ such that $ex \in Ex$ with $\underline{\gamma}(ex, \Phi) = \mathbf{tt}$ and $[\cdot, \phi]_{\underline{\gamma}} = \mathbf{tt}$ for all other nodes on the path.

By the same argument as in Case II.2.1, assumption (2) implies that n is initially added to Sat_{opt} and never removed. On the other hand by the same inductive argument as in Case II.2.2, assumption (1) implies that n is removed from Sat_{pes} after finitely many iterations of the while-loop in the pes-

simistic run. This means we set $\underline{\partial}'(n, \Phi) = \mathbf{??}$ in Line 3 and never modify it afterward, which finally ensures that $\underline{\partial}'(n, \Phi) = [n \models \Phi]_{\underline{\gamma}}$ if $\Phi = \exists \mathsf{G} \phi_1$.

Case III: $\Phi = \exists \phi_1 \mathsf{U} \phi_2$.

The argumentation for this case is at many places very similar to Case II. As such, we only go briefly over the different cases and highlight the differences.

Case III.1: $n \in Ex$.

For this case, the argument is similar as for the analogous case where $\Phi = \exists \mathsf{G} \phi_1$. The difference is that for $\underline{\gamma}(n, \Phi) = \mathbf{ff}$ we see n is not initially added to Sat_{pes} or Sat_{opt}. Since we add states to Sat_{pes} and Sat_{opt} in the respective while-loops, we need to ensure n is never added. This however trivially holds, since n being an exit node implies it does not have any successors and thus will never indeed not be added to Sat_{pes} or Sat_{opt}.

Case III.2: $n \notin Ex$.

Case III.2.1: $[n \models \Phi]_{\underline{\gamma}} = \mathbf{tt}$.

Here, we must have a path $n \longrightarrow_+ n_1 \ldots \longrightarrow_+ n_l$ such that either $[n_l, \phi_2]_{\underline{\gamma}} = \mathbf{tt}$ or $n_l \in Ex$ and $\underline{\gamma}(n_l, \Phi) = \mathbf{tt}$, and additionally $[\cdot, \phi_1]_{\underline{\gamma}} = \mathbf{tt}$ for all other nodes on the path.

Similar to Case II.2.2 we can inductively prove that after finitely many iterations of the respective while-loop, n is an element of both Sat_{pes} and Sat_{opt}. For this, first notice that n_l is in both Sat_{pes} and Sat_{opt} initially. After m iterations of the respective while-loop we can in the same inductive fashion as before prove that n_{l-m} is in Sat_{pes} and Sat_{opt}. In particular this implies that $n \in Sat_{pes} \cap Sat_{opt}$ after l iterations. Hence, we set $\underline{\partial}(n, \Phi) = \mathbf{tt}$.

Case III.2.2: $[n \models \Phi]_{\underline{\gamma}} = \mathbf{ff}$.

Toward a contradiction, assume that $n \in Sat_{opt}$ after the while-loop. As $[n \models \Phi]_{\underline{\gamma}} = \mathbf{ff}$ we must have $[n \models \phi_2]_{\underline{\gamma}} = \mathbf{ff}$. This implies that n was not added to Sat_{opt} initially in Line 28, but after finitely many iterations of the while-loop. Hence, we must have $[n \models \phi_1]_{\underline{\gamma}} \neq \mathbf{ff}$ there must have been an $n_1 \in Sat_{opt}$ in the previous iteration with $n \longrightarrow_+ n_1$. For $n_1 \in Sat_{opt}$ we must have that it was added to Sat_{opt} initially in Line 28, or $[n_1, \phi_1]_{\underline{\gamma}} \neq \mathbf{ff}$ and n_1 itself has a successor that was in Sat_{opt} in the previous iteration. Inductively, we can see that this implies the existence of a finite path $n \longrightarrow_+ n_1 \ldots \longrightarrow_+ n_l$ such that n_l was initially added to Sat_{opt} in Line 28 and $[\cdot, \phi_1]_{\underline{\gamma}} \neq \mathbf{ff}$. But n_l being in Sat_{opt} initially implies that either $[n_l, \phi_2]_{\underline{\gamma}} \neq \mathbf{ff}$ or $n_l \in Ex$ with $\underline{\gamma}(n_l, \Phi) \neq \mathbf{ff}$. I either case, this is a contradiction to the assumption $[n \models \Phi]_{\underline{\gamma}} = \mathbf{ff}$.

Hence, we showed $n \notin Sat_{opt}$ after the while-loop and thus $\underline{\partial}(n, \phi) = \mathbf{ff}$.

Case III.2.3: $[n \models \Phi]_{\underline{\gamma}} = \mathbf{??}$.

Analogously to the argument for $\exists \mathsf{G} \phi_1$, by combining the arguments from Cases III.2.1 and III.2.2 we can deduce that $\underline{\partial}(n, \Phi) = \mathbf{??}$. □

A.2 Contextualization

Lemma 4 *Let $\underline{\mathcal{A}}$ be an RSM with consistent contexts $\underline{\gamma}$ and consistent interpretation $\underline{\partial}$ over a CTL formula Φ. Then for any input box b Algorithm 2 returns $\underline{\gamma}'$ and $\underline{\partial}'$ that are consistent with $\underline{\mathcal{A}}'$.*

Proof Let $[\![\underline{\mathcal{A}}]\!] = (B^* \times N, \Longrightarrow, AP, L)$ be the underlying Kripke structure of $\underline{\mathcal{A}}$ and $\sigma b \in B^*$ a call stack. From the Kripke structure semantics, in particular rules (loop) and (return) (cf. Figure 2) it follows that for all box stacks $\sigma \in B^*$ the states $(\sigma b, ex)$ and $\big(\sigma, (b, ex)\big)$ in $[\![\underline{\mathcal{A}}]\!]$ have the same successors. Further, by definition of the labeling function L, they are labeled by the same atomic propositions. Thus, they must satisfy the same CTL formulas.

This means that if we redefine $Y_i(b)$ such that $\gamma_{Y_i(b)}$ agrees with $\underline{\partial}$ on all exit nodes of $\mathcal{A}_{Y_i(b)}$ we have by consistency of $\underline{\partial}$ that $\gamma_{Y_i(b)}$ remains consistent as well.

In case we introduce a new component, we can apply the same reasoning to ensure that the new context γ_{k+1} is consistent and thus $\underline{\gamma}$ remains consistent. What remains to be shown is that $\underline{\partial}$ stays consistent. For this, assume that for some $\phi \in Subf(\Phi)$ we have $\underline{\partial}'(n', \phi) = \mathbf{tt}$. This means that $\underline{\partial}(n, \phi) = \mathbf{tt}$. By consistency of $\underline{\partial}$ we must have $(\sigma, n) \models \phi$ for all box stacks $\sigma \in B^*$. Since $\mathcal{A}_{k+1}$ and $\mathcal{A}_{Y_i(b)}$ are isomorphic, and we only replaced $Y_i(b)$ by $k+1$ this means there is an homomorphism from $[\![\underline{\mathcal{A}}']\!]$ to $[\![\underline{\mathcal{A}}]\!]$, where the homomorphism simply maps all states (σ', n') to (σ, n) and states (σ', n) to (σ, n) where σ is obtained by replacing all occurrences of b_i' by b_i for all $b_i' \in B_{k+1}$. Thus, from $(\sigma, n) \models \phi$ in $[\![\underline{\mathcal{A}}]\!]$ for all box stacks $\sigma \in B^*$ it follows that $(\sigma, n) \models \phi$ in $[\![\underline{\mathcal{A}}']\!]$ for all box stacks $\sigma \in B'^*$ and hence $\underline{\partial}'(n', \phi) = \mathbf{tt}$ does not violate consistency. The same argumentation applies when $\underline{\partial}(n, \phi) = \mathbf{ff}$. In case $\underline{\partial}(n, \phi) = \mathbf{??}$ we have $\underline{\partial}'(n', \phi) = \mathbf{??}$ which trivially cannot violate consistency.

Finally, removing unreachable components does not have any influence on consistency of the other components. Therefore, $\underline{\partial}'$ is consistent. □

Lemma 5 *For an RMS $\underline{\mathcal{A}}$ and CTL formula Φ Algorithm 3 returns an RSM $\underline{\mathcal{A}}'$ with consistent contexts $\underline{\gamma}'$.*

Proof The initialization performed by INITIALIZE can be seen as contextualizing the initial component $\mathcal{A}_1$ such that the rule (loop) of the semantics of the RSM $[\![\underline{\mathcal{A}}]\!]$ is respected. By applying the same reasoning steps as in the proof of Lemma 4 we can see that $\underline{\gamma}'$ is consistent. □

A.3 Eager RSM model checking

Lemma 6 *Let $\underline{\mathcal{A}}$ be an RSM with consistent contexts $\underline{\gamma}$ and consistent interpretation $\underline{\partial}$ over a formula Φ. If $\underline{\mathcal{A}}$ has been initialized via* INITIALIZE, *and $\underline{\partial}$ =* REFINETERNARY

*$(\underline{\mathcal{A}}, \Phi, \underline{\partial}, \underline{\gamma})$ and $\underline{\mathcal{A}}, \underline{\gamma}, \underline{\partial} = \textsc{Contextualize}(\underline{\mathcal{A}}, \Phi, \underline{\gamma}, \underline{\partial}, b)$ for all $b \in B$, i.e., both are fixed points, and $\underline{\partial}(n, \phi) \neq$ **??** $\neq \underline{\gamma}(n, \phi)$ for all $n \in \mathsf{N}^{all}$ and $\phi \in Subf(\Phi)\backslash\{\Phi\}$, then $\underline{\partial}(n, \Phi) =$ **??** implies that either*

- *$\Phi = \exists\mathsf{G}\phi_1$ and $(\sigma, n) \models \Phi$ for all $\sigma \in B^*$, or*
- *$\Phi = \exists\phi_1 \mathsf{U} \phi_2$ and $(\sigma, n) \not\models \Phi$ for all $\sigma \in B^*$.*

Proof First, notice that under the given assumptions $\underline{\partial}(n, \Phi) =$ **??** implies that Φ is either an $\exists\mathsf{G}$- or an $\exists\mathsf{U}$-formula. This is because $\Phi = a$ for $a \in AP$ then trivially $\underline{\partial}(n, \Phi) \neq$ **??** by RefineTernary. Similarly, $\underline{\partial}(n, \phi) \neq$ **??** for all strict subformulas ϕ of Φ implies that $\underline{\partial}(n, \Phi) \neq$ **??** for Φ being a propositional formula, i.e., negation or disjunction. Lastly, if $\Phi\exists\mathsf{X}\phi_1$ we must have $\underline{\partial}(n, \Phi) \neq$ **??**, which follows from either $\underline{\partial}(n, \phi) \neq$ **??** (for $n \notin Ex$), or $\underline{\gamma}(n, \phi) \neq$ **??** (for $n \in Ex$). This leaves only $\exists\mathsf{G}$- or an $\exists\mathsf{U}$-formulas as possibilities for Φ.

First, we consider the case $\Phi = \exists\mathsf{G}\phi_1$ and assume for an arbitrary node $n \in \mathsf{N}^{all}$ that $\underline{\partial}(n, \Phi) =$ **??** for $\underline{\partial}$ a fixed point of RefineTernary and Contextualize. As $\underline{\partial}(n, \phi_1) \neq$ **??**, the pessimistic and optimistic runs in RefineTernary only differ in the context-dependent part of the initialization, i.e., Line 15 and Line 19 as $\underline{\partial}(n, \phi_1) \neq$ **ff** is equivalent to $\underline{\partial}(n, \phi_1) =$ **tt**. However, $\underline{\partial}(n, \Phi) =$ **??** implies that $n \in Sat_{opt}$ but $n \notin Sat_{pes}$. Thus there must be an exit node $ex \in Sat_{opt}$ but $ex \notin Sat_{pes}$ that is reachable from n through a path $n \longrightarrow_+ \ldots \longrightarrow_+ ex$ on which $\underline{\partial}(\cdot, \phi_1) =$ **tt** and in which $\underline{\gamma}(ex, \Phi) =$ **??**. As Contextualize is a fixed point as well, we must have that $\underline{\partial}((b, ex), \Phi) = \underline{\gamma}(ex, \Phi) =$ **??** for all $b \in B$. Since $(b, ex) \in \mathsf{N}^{all}$ we can apply the same reasoning as we did for n and find a path $(b, ex) \longrightarrow_+ \ldots \longrightarrow_+ ex'$ to an exit node $ex' \in Ex$. Inductively applying this reasoning we can construct an infinite (possibly cyclic) path in the underlying Kripke structure $[\![\underline{\mathcal{A}}]\!]$:

$$
\begin{aligned}
(\sigma_0, n) &\Longrightarrow \ldots \Longrightarrow (\sigma_1 b, ex) \Longrightarrow \\
(\sigma_1, (b, ex)) &\Longrightarrow \ldots \Longrightarrow (\sigma_2 b', ex') \Longrightarrow \\
(\sigma_2, (b', ex')) &\Longrightarrow \ldots
\end{aligned}
$$

The existence of a path from (σ_0, n) to $(\sigma_1 b, ex)$ is implied by the $\longrightarrow_+$-path from n to ex. Notice that b here is arbitrary and by writing the box stack in ex as $\sigma_1 b$ we only assume the existence of such a box b, i.e., that the box stack is not empty in ex. Notice that indeed the box stack in ex cannot be empty since an empty box stack would imply $\underline{\gamma}(ex, \Phi) \neq$ **??** as only the initial component can be reached with the empty box stack and the initial component has a fully specified context (i.e., does not map to **??**) by definition of Initialize. The transition from $(\sigma_1 b, ex) \Longrightarrow (\sigma_1, (b, ex))$ exists by definition of $[\![\underline{\mathcal{A}}]\!]$, specifically rule (exit). Lastly, notice that $(\sigma, n) \models \phi_1$ for all states (σ, n) on the constructed path in $[\![\underline{\mathcal{A}}]\!]$. For $n \notin Ex$ this immediately follows from $\underline{\partial}(n, \phi_1) =$ **tt** as discussed before and consistency of $\underline{\partial}$. Thus, the constructed path is a witness for $(\sigma, n) \models \Phi$.

Next, consider the case $\Phi = \exists\phi_1 \mathsf{U} \phi_2$. Toward a contradiction, assume $(\sigma, n) \models \Phi$ for all $\sigma \in B^*$. This means there must be a finite path

$$(\sigma, n) \Longrightarrow \ldots \Longrightarrow (\sigma', n')$$

such that $(\sigma', n') \models \phi_2$ and for all other states s along the path $s \models \phi_1$. By consistency of $\underline{\partial}$ and the fact that $\underline{\partial}(\cdot, \cdot) \neq$ **??** for strict subformulas of Φ it follows that $\underline{\partial}(n', \phi_2) =$ **tt** and $\underline{\partial}(m, \phi_1) =$ **tt** for all other states $(\cdot, m)$ on the path. If there was a path $n \longrightarrow_+ n'$ it would immediately follow from by Lemma 3 and definition of the ternary semantics that $\underline{\partial}(n, \Phi) =$ **tt** which would already be a contradiction. Thus, there cannot be a path $n \longrightarrow_+ n'$ which necessarily means that the path in $[\![\underline{\mathcal{A}}]\!]$ must contain at least one exit node. Let ex be the last return node (b, ex) on the path such that $ex \in Ex$. Then $(b, ex) \longrightarrow_+ \ldots \longrightarrow_+ n'$ and hence again by Lemma 3 and definition of the ternary semantics we have $\underline{\partial}((b, ex), \Phi) =$ **tt**. As Contextualize is a fixed point, we also have $\underline{\gamma}(ex, \Phi) = \underline{\partial}((b, ex), \Phi) =$ **tt**. Now we can apply the same reasoning to find that the statement also holds for the last return node (b', ex') on the partial path $(\sigma, n) \Longrightarrow \ldots \Longrightarrow (\sigma_1 b, ex)$ and hence also $\underline{\partial}(ex', \Phi) =$ **tt**. Inductively applying this argument and using the fact that the path is finite (thus of course containing finitely many states of exit nodes), we can see that $\underline{\partial}(ex_i, \Phi) =$ **tt** holds for all exit nodes $ex_i \in Ex$ such that $(\cdot, ex_i)$ appears on the path, in particular for the exit node ex_1 for which a state $(\cdot, ex_1)$ first appears on the path. Thus, all transitions on the path $(\sigma, n) \Longrightarrow \ldots \Longrightarrow (\cdot, ex_1)$ must be obtained from rules (loc) or (call) in the Kripke structure semantics. Hence, $n \longrightarrow_+ \ldots \longrightarrow_+ ex_1$. But then by $\underline{\partial}(ex_1, \Phi) = \underline{\gamma}(ex_1, \Phi) =$ **tt** and Lemma 3 along with definition of the ternary semantics we have $\underline{\partial}(n, \Phi) =$ **tt**. But this contradicts our assumption $\underline{\partial}(n, \Phi) =$ **??**. Hence $(\sigma, n) \not\models \Phi$. □

Theorem 1 *Algorithm 4 terminates and is correct, i.e., returns **tt** iff $\underline{\mathcal{A}} \models \Phi$ and **ff** iff $\underline{\mathcal{A}} \not\models \Phi$ for any RSM $\underline{\mathcal{A}}$ and CTL formula Φ over a common set of atomic propositions.*

Proof We first show termination. For this, we show that RefineTernary and Contextualize reach a fixed point after finitely many iterations. First, notice that Contextualize can only create finitely many new components. This is because whenever Contextualize does create a new component $\mathcal{A}_{k+1}$, its context γ_b induced by the box b that was given to Contextualize as parameter is a strict refinement of the component $\mathcal{A}_{Y(b)}$ to which b redirected before Contextualize was called, i.e., γ_b is a refinement of $\gamma_{Y(b)}$ and there is an exit node ex for which $\gamma_{Y(b)} =$ **??** but $\gamma_b \neq$ **??**. Hence, any box b can only be refined finitely

many times. Notice that the number of boxes is not constant as it may also increase when a copy of a component is created, but it is always bounded by the number of copies of a component that can be created, which in turn in finite as it is bounded by the number of contexts. This is sufficient to ensure CONTEXTUALIZE eventually reaches a fixed point. Similarly, notice that if $\underline{\gamma}'$ is a refinement of $\underline{\gamma}$, then the ternary interpretation induced by the ternary semantics $\underline{\partial}' = [n \models \phi]_{\underline{\gamma}}$ is a refinement of $\underline{\partial} = [n \models \phi]_{\underline{\gamma}}$. This directly follows from the definition of the ternary semantics. Since CONTEXTUALIZE only refines contexts, this means as long as CONTEXTUALIZE does not create a new context, the number of node-formula pairs such that $\underline{\partial}(\cdot,\cdot) = \mathbf{??}$ for the ternary interpretation computed by REFINETERNARY monotonically decreases. As again, CONTEXTUALIZE can only create finitely many new components, REFINETERNARY must reach a fixed point.

For the correctness of the results we first show by induction that after executing the loop from Line 2 to Line 14 for a formula $\phi \in Subf(\Phi)$, we have that $\underline{\partial}(n,\phi) \neq \mathbf{??} \neq \underline{\gamma}(ex,\phi)$ for all $n \in \mathsf{N}^{all}$ and $ex \in Ex$. For the base case, where ϕ is an atomic proposition this trivially follows from the definition of $/ALocDed$. If ϕ is an $\exists\mathsf{G}$ or $\exists\,\mathsf{U}$ formula this trivially holds due to the assignments following Line 8. Otherwise, this is proven by application of Lemma 6 where the assumptions hold due to the induction hypothesis.

This means that to show correctness, it is sufficient to show that $\underline{\partial}$ is consistent. The assignments following Line 8 preserve consistency of $\underline{\partial}$ and $\underline{\gamma}$ by Lemma 6. For CONTEXTUALIZE we ensure via Lemma 4 that $\underline{\gamma}$ remains consistent, given that $\underline{\partial}$ is consistent. For consistency of $\underline{\partial}$, first notice that, as shown above, REFINETERNARY being called for formula ϕ implies that for all nodes $n \in \mathsf{N}^{all}$, exit nodes $ex \in Ex$ and strict subformulas ψ of ϕ both $\underline{\partial}(n,\psi)$ and $\underline{\gamma}(ex,\psi)$ are not **??**. By Lemma 2 we have that also $[n \models \psi]_{\underline{\gamma}} \neq \mathbf{??}$ Also by consistency of the ternary semantics (cf. Lemma 1) and assuming consistency of $\underline{\gamma}$, applying Lemma 3 inductively on the subformulas of ϕ we see that $\underline{\partial}$ also actually matches the ternary semantics and is thus also consistent for $\underline{\partial}(\cdot,\phi)$.

This creates a cyclic dependency where $\underline{\partial}$ remains consistent if $\underline{\gamma}$ is consistent, and vice versa. As by Lemma 5 INITIALIZE returns an RSM $\underline{\mathcal{A}}$ with consistent contexts, $\underline{\gamma}$ is initially consistent. Also initially $\underline{\partial}(\cdot,\cdot) = \mathbf{??}$ for all node-formula pairs, which is consistent by definition. Thus, we obtain that $\underline{\gamma}$ and $\underline{\partial}$ remain consistent throughout the algorithm and hence yield correctness of the output result. □

B Lazy RSM model checking

Theorem 2 *Algorithm 5 terminates and is correct for complete expansion heuristic H, i.e.,* LAZYCHECK$(\underline{\mathcal{A}}, \Phi, H)$ *returns **tt** iff $\underline{\mathcal{A}} \models \Phi$ and **ff** iff $\underline{\mathcal{A}} \not\models \Phi$ for any RSM $\underline{\mathcal{A}}$, and CTL formula Φ over a common set of atomic propositions.*

Proof The argument for termination is exactly as in Theorem 1, essentially showing there are only finitely many contexts and for each context there are only finitely many possible refinement steps.

Similarly, consistency of $\underline{\gamma}$ and $\underline{\partial}$ is preserved through subroutines INITIALIZE, REFINETERNARY, CONTEXTUALIZE is given by Lemma 5, Lemma 3 with Lemma 1 and Lemma 2, and Lemma 4, respectively. Also the assignments following Line 6 preserve consistency by Lemma 6. To complete the proof, we show that Lemma 6 is applicable, i.e., all assumptions hold.

Notice that the condition that the truth value for all subformulas is known in every node is explicitly checked. Further, when reaching Line 6, we have $B = \emptyset$. Together with the fact that H is complete this implies that $C(\mathcal{A}, \Phi, \underline{\gamma}, \underline{\partial}) = \emptyset$, i.e., no boxes are contextualizable and hence CONTEXTUALIZE is a fixed point for all $b \in B$. Thus, CONTEXTUALIZE is a fixed point. Lastly, if $\underline{\partial}$ is computed bottom-up for all $\phi \in Subf(\Phi)$ via REFINETERNARY then by Lemma 3 corresponds exactly to the three valued semantics for all node-formula pairs which is uniquely defined and thus REFINETERNARY is a fixed point as well. □

Theorem 3 *Algorithm 5 terminates and is correct for the special expansion heuristic* GETNEXTEXPANSION *with the modified global dependency cycle resolution, i.e.,* LAZYCHECK$'(\underline{\mathcal{A}}, \Phi,$ GETNEXTEXPANSION$)$ *returns **tt** iff $\underline{\mathcal{A}} \models \Phi$ and **ff** iff $\underline{\mathcal{A}} \not\models \Phi$ for any RSM $\underline{\mathcal{A}}$ and CTL formula Φ over a common set of atomic propositions.*

Proof We first show correctness of the approach. As in the proof of Theorem 2 we have already seen that all subroutines preserve consistency of $\underline{\partial}$ and $\underline{\gamma}$, and initially both are consistent by Lemma 5. Thus, we only need to show that the modified cycle dependency resolution modifies $\underline{\gamma}$ and $\underline{\partial}$ in such a way that they remain consistent.

First, consider the case where $\phi = \exists\mathsf{G}\phi_1$ and the if-branch in Line 6 is entered, setting $\underline{\partial}(n_j,\phi) = \mathbf{tt}$ for some node $n_j \in \mathsf{N}^{all}$. Due to the modified global cycle dependency resolution described above this means we have $(n_j,\phi) \in D_{all}$ as well as $\underline{\partial}(n_j,\phi') \neq \mathbf{??}$ for all $\phi' \in Subf(\phi) \setminus \{\phi\}$. Because $(n_j,\phi) \in D_{all}$ we know it must have been added to D in Line 22. Therefore there must be a sequence of node-formula pairs $(n_1,\psi_1)\ldots(n_j,\psi_j)\ldots(n_l,\psi_l)$ where $\psi_j = \phi_1$ such that FINDREASON was recursively called with (n_i,ψ_i) in the input. Additionally, there must be a transition $n_l \longrightarrow n_1$ and we must have $\phi_1 = \phi_l$, such

that FINDREASON$(\cdot, n_l, \phi_l, \cdot, \cdot, \cdot, \cdot)$ went into the if-branch in Line 22 because $(n_1, \phi_1) \in R$ since FINDREASON was called on (n_1, ϕ_1) before. As recursive calls of FINDREASON only happen on formulas monotonically decreasing in size, we have that $\phi_i = \phi$ for all $i \in \{1, \ldots, l\}$. This implies that the if-statement in Line 19 was never entered, and hence $\underline{\partial}(n_i, \phi_1) \neq$ **??**. If, on the other hand, we had $\underline{\partial}(n_i, \phi_1) =$ **ff** for some i, then we would have deduced $\underline{\partial}(n_i, \Phi_1) =$ **ff** by REFINETERNARY and hence FINDREASON$(\cdot, n_i, \phi_i, \cdot, \cdot, \cdot, \cdot)$ would not have been called. Hence, we must have $\underline{\partial}(n_i, \phi_1) =$ **tt** for all $i \in \{1, \ldots, l\}$. Due to FINDREASON being called on each pair in the sequence $(n_1, \phi_1) \ldots (n_j, \phi_1) \ldots (n_l, \phi_1)$, we must have executed either Line 7, Line 12, or Line 24 either of which implies that there is a transition from n_i to n_{i+1}. Thus, we have a cycle starting in n_1 that goes through all n_i and for which $\underline{\partial}(n_i, \phi_1) =$ **tt** for all $i \in \{1, \ldots, l\}$. Due to consistency of $\underline{\partial}$ this is a witness for that $\underline{\partial}(n_i, \Phi) =$ **tt** maintains consistency.

Now we prove consistency of the global dependency cycle resolution for the case where $\phi = \exists \phi_1 \mathsf{U} \phi_2$. Toward a contradiction, assume that we set $\underline{\partial}(n_1, \phi) =$ **ff** in the global dependency cycle resolution despite there being a finite shortest witnessing path for ϕ of length l. W.l.o.g. we call the nodes on this path n_i for $i \in \{1, \ldots, l\}$. Since the global dependency cycle resolution assures that all subformulas of ϕ are known in all nodes, we must have $\underline{\partial}(n_l, \phi_2) =$ **tt**, and $\underline{\partial}(n_i, \phi_1) =$ **tt** and $\underline{\partial}(n_i, \phi_2) =$ **ff** for $i \in \{1, \ldots, l-1\}$. Also REFINETERNARY ensures that $\underline{\partial}(n_l, \phi_1) =$ **tt**. However, since $\underline{\partial}(n_1, \phi) =$ **??** before the global dependency cycle resolution there must be a largest j such that $\underline{\partial}(n_j, \phi) =$ **??** but $\underline{\partial}(n_{j+1}, \phi) =$ **tt**. If $n_j \longrightarrow_+ n_{j+1}$ we would have that REFINETERNARY would have set $\underline{\partial}(n_j, \phi) =$ **tt**. Hence we do not have $n_j \longrightarrow_+ n_{j+1}$. Since the witnessing path contains n_j followed by n_{j+1} and $\longrightarrow_+$ covers all local transitions as well as transitions entering boxes, this leaves only one possibility for n_j and n_{j+1}, namely that $n_j \in Ex$ and $n_{j+1} = (b, n_j)$ for some $b \in B$. However then we would have returned the non-empty set $\{b\}$ along with D in Line 9 in the recursive call of FINDREASON.

Toward showing termination, notice first that FINDREASON can only call itself recursively finitely many times, as there are only finitely many possible node-formula pairs. Because whenever we call FINDREASON with a node-formula pair (n, ϕ) in the input we immediately add (n, ϕ) to R, the recursion can only have finite depth. Because the RSM and the formula are finite we can also immediately see that the width of the recursion tree is finite. Thus, FINDREASON terminates. Lastly, because FINDREASON always returns a pair $\{b\}, D$ where either $b \in C(\underline{\mathcal{A}}, \underline{\partial})$ or $D \neq \emptyset$, we always either add contextual information to a box or perform a global dependency cycle resolution. Since the former can only be done finitely many times and the latter strictly reduces the number of node-formula pairs for which $\underline{\partial}(\cdot, \cdot) =$ **??**, we eventually have $\underline{\partial}(en_1, \Phi) \neq$ **??** for all $en_1 \in En_1$ which means the algorithm terminates. □

References

1. The Apache™ FOP Project. https://xmlgraphics.apache.org/fop/. Accessed 31 Jan 2024 (2016)
2. The AVR Simulation and Analysis Framework . https://github.com/avrora-framework/avrora. Accessed 31 Jan 2024 (2016)
3. 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 (2005)
4. Alur, R., Bouajjani, A., Esparza, J.: Model Checking Procedural Programs, pp. 541–572. Springer, Cham (2018)
5. Alur, R., Yannakakis, M.: Model checking of hierarchical state machines. ACM Trans. Program. Lang. Syst. **23**(3), 273–303 (2001)
6. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)
7. Bernasconi, A., Menghi, C., Spoletini, P., Zuck, L.D., Ghezzi, C.: From model checking to a temporal proof for partial models. In: Cimatti, A., Sirjani, M. (eds.) Software Engineering and Formal Methods, pp. 54–69. Springer, Cham (2017)
8. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability—Second Edition, Volume 336 of Frontiers in Artificial Intelligence and Applications. IOS Press (2021)
9. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Mazurkiewicz, A.W., Winkowski, J. (eds.) Proceedings of the CONCUR'97, Volume 1243 of LNCS, pp. 135–150. Springer (1997)
10. Brauer, J., Huuck, R., Schlich, B.: Interprocedural pointer analysis in goanna. Electron. Notes Theor. Comput. Sci. **254**, 65–83 (2009). Proceedings of the 4th International Workshop on Systems Software Verification (SSV 2009)
11. Brázdil, T.: Verification of Probabilistic Recursive Sequential Programs. Ph.D. thesis, Masaryk University Brno (2007)
12. Bruns, G., Godefroid, P.: Model checking partial state spaces with 3-valued temporal logics. In: Halbwachs, N., Peled, D.A. (eds.) Proceedings of the CAV'99, pp. 274–287. Springer (1999)
13. Bruns, G., Godefroid, P.: Generalized model checking: reasoning about partial state spaces. In: Palamidessi, C. (ed.) CONCUR 2000—Concurrency Theory, pp. 168–182. Springer, Berlin (2000)
14. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **35**, 677–691 (1986)
15. Burkart, O., Steffen, B.: Model checking for context-free processes. In Cleaveland, W. (ed.) Proceedings of the CONCUR'92, pp. 123–137 (1992)
16. Burkart, O., Steffen, B.: Model checking the full modal mu-calculus for infinite sequential processes. Theor. Comput. Sci. **221**(1–2), 251–270 (1999)
17. Chechik, M., Devereux, B., Easterbrook, S., Gurfinkel, A.: Multi-valued symbolic model-checking. ACM Trans. Softw. Eng. Methodol. **12**(4), 371–408 (2003)
18. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Logic of Programs, Volume 131 of LNCS, pp. 52–71 (1981)
19. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification, pp. 154–169. Springer, London (2000)
20. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (2000)

21. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R.: Handbook of Model Checking. Incorporated, 1st edn. Springer, New York (2018)
22. Cousot, P.: Principles of Abstract Interpretation. The MIT Press, Cambridge (2021)
23. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Los Angeles, California, pp. 238–252. ACM Press, New York (1977)
24. Dams, D., Grumberg, O.: Abstraction and Abstraction Refinement, pp. 385–419. Springer, Cham (2018)
25. Dubslaff, C., Wienhöft, P., Fehnker, A.: Be lazy and don't care: faster CTL model checking for recursive state machines. In: Calinescu, R., Păsăreanu, C.S. (eds.) Software Engineering and Formal Methods, pp. 332–350. Springer, Cham (2021)
26. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Engineering, ICSE '99, pp. 411–420. Association for Computing Machinery, New York (1999)
27. Etessami, K., Yannakakis, M.: Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations. J. ACM **56**(1), 1–66 (2009)
28. Fehnker, A., Dubslaff, C.: Inter-procedural analysis of computer programs. US Patent 8,296,735 (2012)
29. Gmytrasiewicz, P.J., Durfee, E.H.: A logic of knowledge and belief for recursive modeling: a preliminary report. In: Swartout, W.R. (ed.) Proceedings of the 10th National Conference on Artificial Intelligence, San Jose, CA, USA, July 12–16, 1992, pp. 628–634. AAAI Press/The MIT Press (1992)
30. Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-based model checking using modal transition systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001—Concurrency Theory, pp. 426–440. Springer, Berlin (2001)
31. Göller, S., Lohrey, M.: Branching-time model checking of one-counter processes and timed automata. SIAM J. Comput. **42**(3), 884–923 (2013)
32. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) Proceedings of the Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22–25, 1997, Volume 1254 of Lecture Notes in Computer Science, pp. 72–83. Springer (1997)
33. Hague, M., Ong, C.-H.: A saturation method for the modal μ-calculus over pushdown systems. Inf. Comput. **209**(5), 799–821 (2011)
34. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '02, pp. 58–70, New York, NY, USA. Association for Computing Machinery (2002)
35. Horwitz, S., Reps, T., Sagiv, M.: Demand interprocedural dataflow analysis. In: Kaiser, G.E. (ed.) Proceedings of the SIGSOFT'95, pp. 104–115. ACM (1995)
36. Huth, M.: Model checking modal transition systems using Kripke structures. In: Cortesi, A. (ed.) Verification, Model Checking, and Abstract Interpretation, pp. 302–316. Springer, Berlin (2002)
37. Huth, M., Jagadeesan, R., Schmidt, D.: Modal transition systems: a foundation for three-valued program analysis. In: Sands, D. (ed.) Programming Languages and Systems, pp. 155–169. Springer, Berlin (2001)
38. Jensen, S.H., Møller, A., Thiemann, P.: Interprocedural analysis with lazy propagation. In: Cousot, R., Martel, M. (eds.) Static Analysis, pp. 320–339. Springer, Berlin (2010)
39. Kozen, D.: Results on the propositional μ-calculus. Theor. Comput. Sci. **27**(3), 333–354 (1983). Special Issue Ninth International Colloquium on Automata, Languages and Programming (ICALP) Aarhus, Summer 1982
40. Křetínský, J.: 30 Years of Modal Transition Systems: Survey of Extensions and Analysis, pp. 36–74. Springer, Cham (2017)
41. Larsen, K., Thomsen, B.: A modal process logic. In: [1988] Proceedings. Third Annual Symposium on Logic in Computer Science, pp. 203–210 (1988)
42. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) Computer Aided Verification, pp. 123–136. Springer, Berlin (2006)
43. Menghi, C., Rizzi, A.M., Bernasconi, A., Spoletini, P.: Torpedo: witnessing model correctness with topological proofs. Form. Asp. Comput. **33**(6), 1039–1066 (2021)
44. Menghi, C., Spoletini, P., Ghezzi, C.: Dealing with incompleteness in automata-based model checking. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016: Formal Methods, pp. 531–550. Springer, Cham (2016)
45. 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)
46. Schmidt, D., Steffen, B.: Program analysis as model checking of abstract interpretations. In: Levi, G. (ed.) Static Analysis, pp. 351–380. Springer, Berlin (1998)
47. Schwoon, S.: Model checking pushdown systems. Ph.D. thesis, Technical University Munich, Germany (2002)
48. Sharir, M., Pnueli, A.: Two Approaches to Interprocedural Data Flow analysis, Chapter 7, pp. 189–234. Prentice-Hall, Englewood Cliffs (1981)
49. Song, F., Touili, T.: PuMoC: a CTL model-checker for sequential programs. In: Goedicke, M., Menzies, T., Saeki, M. (eds.) Proceedings of the ASE'12, pp. 346–349. ACM (2012)
50. Uchitel, S., Alrajeh, D., Ben-David, S., Braberman, V., Chechik, M., De Caso, G., D'Ippolito, N., Fischbein, D., Garbervetsky, D., Kramer, J., Russo, A., Sibay, G.: Supporting incremental behaviour model elaboration. Comput. Sci. Res. Dev. **28**(4), 279–293 (2013)
51. Vizel, Y., Grumberg, O., Shoham, S.: Lazy abstraction and sat-based reachability in hardware model checking. In: 2012 Formal Methods in Computer-Aided Design (FMCAD), pp. 173–181 (2012)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Clemens Dubslaff is an Assistant Professor from the formal system analysis cluster at Eindhoven University of Technology, The Netherlands. He holds an M.Sc. degree in Computational Logic from UN Lisbon, Portugal, and a Ph.D. from Dresden University of Technology, Germany. His research interests cover formal methods and their explainability using symbolic techniques and causal reasoning.

Patrick Wienhöft is a Research Assistant at Dresden University of Technology, Germany. He completed his M.Sc. in Computational Logic with a specialization in temporal logic and verification. Since December 2020 he pursues a Ph.D. at the cluster of excellence CeTI, in which he is conducting research on reinforcement learning and computational models of human behaviors.

Ansgar Fehnker is an Associate Professor (Teaching and Leadership) in the School of Computing at Macquarie University. In his career he held positions at the University of Twente, University of the South Pacific, and National ICT Australia in Sydney. His research interest is the use of automated system and software verification tools, in particular model checking and static analysis, and their application in the design and development of software systems.