Model Checking as Program Verification by Abstract Interpretation

Paolo Baldan
Department of Mathematics, University of Padua, Italy

Roberto Bruni
Department of Computer Science, University of Pisa, Italy

Francesco Ranzato
Department of Mathematics, University of Padua, Italy

Diletta Rigo
Department of Mathematics, University of Padua, Italy

Abstract

Abstract interpretation offers a powerful toolset for static analysis, tackling precision, complexity and state-explosion issues. In the literature, state partitioning abstractions based on (bi)simulation and property-preserving state relations have been successfully applied to abstract model checking. Here, we pursue a different track in which model checking is seen as an instance of program verification. To this purpose, we introduce a suitable language – called MOKA (for MOdel checking as abstract interpretation of Kleene Algebras) – which is used to encode temporal formulae as programs. In particular, we show that (universal fragments of) temporal logics, such as ACTL or, more generally, universal μ-calculus can be transformed into MOKA programs. Such programs return all and only the initial states which violate the formula. By applying abstract interpretation to MOKA programs, we pave the way for reusing more general abstractions than partitions as well as for tuning the precision of the abstraction to remove or avoid false alarms. We show how to perform model checking via a program logic that combines under-approximation and abstract interpretation analysis to avoid false alarms. The notion of locally complete abstraction is used to dynamically improve the analysis precision via counterexample-guided domain refinement.

2012 ACM Subject Classification Theory of computation → Modal and temporal logics; Theory of computation → Verification by model checking; Theory of computation → Program verification; Theory of computation → Programming logic; Theory of computation → Abstraction

Keywords and phrases ACTL, μ-calculus, model checking, abstract interpretation, program analysis, local completeness, abstract interpretation repair, domain refinement, Kleene algebra with tests

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.8

Related Version *Extended Version*: `https://arxiv.org/abs/2506.05525` [1]

Funding This research was partially funded by the *Italian MUR*, under the PRIN 2022 PNRR project no. P2022HXNSC on "Resource Awareness in Programming: Algebra, Rewriting, and Analysis", and the PNRR project *SEcurity and RIghts In the CyberSpace* (SERICS, PE00000014 – CUP H73C2200089001), by the INdAM-GNCS Projects RISICO (CUP E53C22001930001) and MARQ (CUP E53C24001950001), by a *WhatsApp Research Award* and by an *Amazon Research Award* for *AWS Automated Reasoning*.

1 Introduction

Abstraction is a fundamental craft for mastering complexity. In model checking, abstraction-guided space reduction allows to mitigate the well-known state explosion problem [15]. Abstract interpretation [16, 17] is the de facto standard framework for designing sound analyses. The idea of applying abstract interpretation to model checking has been widely

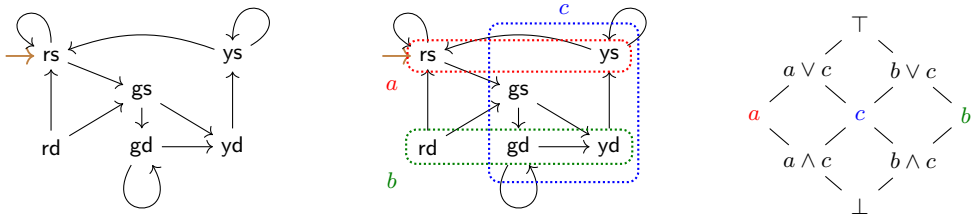
investigated, e.g., [2–5, 10–12, 14, 19–23, 25, 26, 28–30, 33, 35, 37, 38, 40, 44–47, 49, 50, 55] (§ 7 accounts for more closely related work). These approaches often rely on over-approximations that preserve some logical properties, like state partitioning abstractions or simulation-preserving relations.

Objective. In this work, we pursue a different approach, which consists in recasting model checking of temporal formulae directly in terms of program verification, for which the whole tool-suite of abstract interpretation is readily available. To this aim, we exploit an instance of a Kleene algebra with tests and fixpoints (KAF) [36], whose primitives allow to map each temporal formula to a program that can single out all and only the counterexamples to the formula. This program can then be analysed through a sound abstract interpretation that over-approximates the concrete behaviour, so that all possible counterexamples are exposed. However, this over-approximation might not faithfully model the program behaviour in the abstract domain, thus possibly mixing true and false alarms. This lack of precision cannot happen when the abstract interpretation is complete [18]. However, completeness happens quite seldom, and even if, in principle, it can be achieved by refining the abstract domain [27], the most abstract domain refinement ensuring completeness is often way too concrete (it might well coincide with the concrete domain itself).

To remove false alarms, the validity of temporal formulae can be analysed by deriving certain judgements for the corresponding program in a suitable program logic. This enables the use of techniques borrowed from locally complete abstract interpretation [7, 9], where the over-approximation provided by the abstract domain is paired with an under-approximating specification, in the style of O'Hearn's incorrectness logic [43]. This way, any alarm raised by an incorrectness logic proof corresponds to a true counterexample and local completeness guarantees that derivable judgements either exposes some true counterexamples (if any) or proves that there are none. Moreover, the failure of a proof obligation during inference can point out how to dynamically refine the underlying abstraction to enhance the precision and expressiveness of the analysis, a technique called abstract interpretation repair [8].

Methodology (and a toy example). Our main insight is to design a meta-programming language, called MOKA (MOdel checking as abstract interpretation of Kleene Algebras), where suitable temporal formulae can be mapped to. We show how this can be done for ACTL or, more generally, for the single variable μ-calculus without the existential diamond modality. MOKA is a language of regular commands, based on Kleene algebra with tests (KAT) [34] augmented with fixpoint operators (KAF) [36]. MOKA leverages a small set of primitives to extend and filter computation paths. They can be combined with the usual KAF operators of sequential composition, join (i.e., nondeterministic choice), Kleene iteration, and least fixpoint computation. The corresponding programs are then analysed by abstract interpretation.

The language MOKA operates on computation paths $\langle \sigma, \Delta \rangle$, where $\sigma \in \Sigma$ represents the current system state, and $\Delta \in \mathcal{P}(\Sigma)$ represents the set of traversed states. In the following, we informally overload the symbol σ to denote $\langle \sigma, \varnothing \rangle$. Several computation paths can be stacked and unstacked through the MOKA primitives push and pop for dealing with nested temporal formulae. A generic stack is denoted $\langle \sigma, \Delta \rangle :: S$ and we call σ its *current* state. Each temporal formula φ is mapped to a MOKA program $\lfloor \overline{\varphi} \rceil$ that intuitively computes counterexamples to φ, whence the bar over φ in the application of the encoding $\lfloor \cdot \rceil$. Letting $[\![\cdot]\!]$ denote the usual (collecting) denotational semantics, a key result is, therefore, that a state σ satisfies the formula φ iff the execution of $\lfloor \overline{\varphi} \rceil$ on σ returns the empty set. Of course, the results



(a) Behaviour of cars at US traffic light. **(b)** Abstract system. **(c)** Abstract domain.

Figure 1 Variations on the traffic light example from [12].

can be generalised additively to sets of stacks, for which $\lfloor\overline{\varphi}\rceil$ filters out exactly those stacks whose current state satisfies φ. We can thus summarise the semantic correspondence between formulae satisfaction and execution of their MOKA programs by $[\![\lfloor\overline{\varphi}\rceil]\!]P = \{\sigma \in P \mid \sigma \not\models \varphi\}$. As a special case, it follows that $\sigma \models \varphi$ iff $[\![\lfloor\overline{\varphi}\rceil]\!]\{\sigma\} = \varnothing$. For example, for an atomic proposition p, we let $\lfloor\overline{p}\rceil \triangleq \neg\mathsf{p}?$, where $\neg\mathsf{p}?$ is a MOKA primitive that filters out those states where p is valid. As a further example, the MOKA program for the formula $\mathsf{AG}\ \varphi$ (stating that φ along every possible path always holds) is $\lfloor\overline{\mathsf{AG}\ \varphi}\rceil = \mathsf{push}; \mathsf{next}^*; \lfloor\overline{\varphi}\rceil; \mathsf{pop}$.

The intuition is quite simple: next^* generates all the successors by iterating the next operation, and they are filtered by $\lfloor\overline{\varphi}\rceil$, to expose those which fail to satisfy φ. The operations push and pop manage the stack structure. In the example, to attain the semantic correspondence above, whenever a state δ such that $\delta \not\models \varphi$ is reachable from some initial state σ, we want $[\![\lfloor\overline{\mathsf{AG}\ \varphi}\rceil]\!]\{\sigma\}$ to return σ and not δ: the operation push serves to stash σ when looking for δ, and the operation pop to discard δ and unstash σ, to conclude $\sigma \not\models \mathsf{AG}\ \varphi$. To make the next introductory examples easier to follow, we suggest that the reader ignore the push and pop symbols, hence their faded rendering in MOKA programs.

To get a flavour of the proposed approach, we sketch an easy example, which is a variation of [12, Examples 3.4, 3.7]. Consider the transition system in Figure 1a, which models the behaviour of cars at a US traffic light. State names consist of two letters, denoting, respectively, the traffic light status, i.e. red, yellow, green, and the car behaviour, i.e. stop, drive. We model check the validity in the initial state rs of the safety property $\varphi = \mathsf{AG}\ (\neg\mathsf{rd})$, stating that it will never happen that the light is red and the car is driving. This is obviously true for the system, since rd is an unreachable state. We therefore consider the MOKA program $\lfloor\overline{\varphi}\rceil = \mathsf{push}; \mathsf{next}^*; \mathsf{rd}?; \mathsf{pop}$, and compute its semantics $[\![\lfloor\overline{\varphi}\rceil]\!]\{\mathsf{rs}\}$, which turns out to be the empty set, thus allowing us to conclude that $\mathsf{rs} \models \varphi$. In fact, after a few concrete steps of iteration, $[\![\mathsf{rd}?]\!][\![\mathsf{next}^*]\!]\{\mathsf{rs}\} = [\![\mathsf{rd}?]\!]\{\mathsf{rs}, \mathsf{gs}, \mathsf{gd}, \mathsf{yd}, \mathsf{ys}\} = \varnothing$.

How the abstraction works. We show how any sound abstraction of state properties in $\wp(\Sigma)$ of the original system can be lifted to a range of sound abstractions on stacks with varying degrees of precision, in a way that MOKA programs can be analysed by an abstract interpreter, denoted by $[\![\cdot]\!]^\sharp$. The analysis output is always sound, meaning that the set of counterexamples is over-approximated. In particular, letting α denote the abstraction map to an abstract lattice, and $\bot$ the bottom element of the corresponding abstract computational domain, if $[\![\lfloor\overline{\varphi}\rceil]\!]^\sharp\alpha(\{\sigma\}) = \bot$, then $\sigma \models \varphi$ holds. Vice versa, if $[\![\lfloor\overline{\varphi}\rceil]\!]^\sharp\alpha(\{\sigma\}) \neq \bot$, then the abstract analysis might raise a false alarm because, in general, abstractions are not complete.

Back to the previous example, let us consider the (non-partitioning) abstract domain $A \triangleq \{\bot_A, a \wedge c, b \wedge c, a, b, c, a \vee c, b \vee c, \top_A\}$ in Figure 1c, induced by the abstract properties a, b and c, whose concretizations are represented as dotted boxes in Figure 1b. The abstract interpreter computes $[\![\lfloor\overline{\mathsf{AG}\ (\neg\mathsf{rd})}\rceil]\!]^\sharp\alpha(\{\mathsf{rs}\}) = [\![\mathsf{push}; \mathsf{next}^*; \mathsf{rd}?; \mathsf{pop}]\!]^\sharp a = \bot$, thus allowing us

to conclude that $\mathsf{rs} \models \mathsf{AG}\ (\neg\mathsf{rd})$. Since rs has a transition to gs, we have $[\![\mathsf{rd?}]\!]^\sharp [\![\mathsf{next}^*]\!]^\sharp a = [\![\mathsf{rd?}]\!]^\sharp (a \vee c) = \bot$. It is worth remarking that in the abstract domain A, computing $[\![\mathsf{next}^*]\!]^\sharp a$ requires two iterations instead of four, as it happens in the concrete computation. Consider now the formula $\psi = \mathsf{AG}\ (\mathsf{g} \rightarrow \mathsf{AX}\ \mathsf{d})$, namely, "whenever the semaphore is green, then at the next state the car is driving". Spelling out the MOKA encoding, we have: $\lfloor\overline{\psi}\rceil = \mathsf{push}; \mathsf{next}^*; \mathsf{g?}; \mathsf{push}; \mathsf{next}; \neg\mathsf{d?}; \mathsf{pop}; \mathsf{pop}$. Again, ψ holds for the concrete system in Figure 1a, but here the abstract interpretation is imprecise, because $[\![\mathsf{push}; \mathsf{next}^*; \mathsf{g?}]\!]^\sharp \alpha(\{\mathsf{rs}\}) = c::a$ and $[\![\mathsf{push}; \mathsf{next}; \neg\mathsf{d?}]\!]^\sharp (c::a) = (a \vee c)::c::a$, so that $[\![\lfloor\overline{\psi}\rceil]\!]^\sharp \alpha(\{\mathsf{rs}\}) = a \neq \bot$, thus raising a false alarm. As explained above, we exploit push/pop operators to manage the stack structure of domain elements and to recover the (abstract) states from which the computation started when property violations are found. For instance, in the case illustrated above, once the stack $(a \vee c)::c::a$ is obtained, representing an abstract trace where the property fails, its initial abstract state, a, is recovered by applying two successive pop operations.

To eliminate false alarms, we leverage the concept of local completeness in abstract interpretation. Roughly, the idea consists in focusing on the (abstract) computation path produced by some input of interest, and then refining the abstraction only when needed to make it complete *locally* to such computation. More precisely, we apply a variation of local completeness logic (LCL) [7, 9], which is here extended to deal with fixpoint operators. The LCL proof system, parametrised by a generic state abstraction A, works with O'Hearn-like judgements $\vdash_A [P]\ \mathsf{r}\ [Q]$, where r is a program and P, Q denote state properties or, equivalently, the underlying sets of states satisfying those properties. The triple $\vdash_A [P]\ \mathsf{r}\ [Q]$ is valid when Q is an under-approximation of the states reachable by r from P while the abstraction of Q over-approximates such reachable states, and the abstract computation of r on the precondition P is locally complete. Roughly, this can be expressed as $Q \subseteq [\![\mathsf{r}]\!]P \subseteq \gamma \circ \alpha(Q)$. In our setting, where the program $\lfloor\overline{\varphi}\rceil$ associated with a formula φ "returns" all the counterexamples to the validity of φ, an inference of $\vdash_A [P]\ \lfloor\overline{\varphi}\rceil\ [Q]$ shows that $Q \subseteq \{\sigma \in P \mid \sigma \not\models \varphi\} \subseteq \gamma \circ \alpha(Q)$, thus bounding the set of counterexamples to φ in P. Hence, if $Q \neq \varnothing$ then φ does not hold for each $\sigma \in Q \subseteq P$, while φ holds for all states $\sigma \in P \setminus (\gamma \circ \alpha(Q))$. If, instead, $Q = \varnothing$, then $\alpha(Q) = \bot$, and φ holds in the whole P. As LCL derivations cannot succeed with locally incomplete abstractions, if some proof obligation fails, the abstract domain needs to be "fixed" to achieve local completeness. This can be accomplished, e.g., by applying the abstraction repair techniques defined in [8].

For the toy traffic light example, we can derive in LCL the program triple $\vdash_A [\{\mathsf{rs}\}]\ \lfloor\overline{\varphi}\rceil\ [\varnothing]$, that confirms the validity of the formula φ. Vice versa, an attempt to derive the triple $\vdash_A [\{\mathsf{rs}\}]\ \lfloor\overline{\psi}\rceil\ [\varnothing]$ fails because of local incompleteness. Roughly, we can successfully derive $\vdash_A [\{\mathsf{rs}\}]\ \mathsf{next}^*; \mathsf{g?};\ [\{\mathsf{gs}, \mathsf{gd}\}]$, but then the execution of next on $\{\mathsf{gs}, \mathsf{gd}\}$ is not locally complete, because $\alpha([\![\mathsf{next}]\!]\ \{\mathsf{gs}, \mathsf{gd}\}) = \alpha(\{\mathsf{gd}, \mathsf{yd}\}) = b \wedge c$, while $[\![\mathsf{next}]\!]^\sharp \alpha(\{\mathsf{gs}, \mathsf{gd}\}) = [\![\mathsf{next}]\!]^\sharp c = a \vee c$. The abstraction repair procedure of [8] would then lead to refine the abstract domain by adding a new abstract element c_1 to represent the concrete set $\gamma(c_1) = \{\mathsf{gs}, \mathsf{gd}\}$. Since abstract domains must be closed under meets, the addition of c_1 will also entail the addition of $c_1 \wedge b$ such that $\gamma(c_1 \wedge b) = \{\mathsf{gd}\}$. Letting $A_1 \triangleq A \cup \{c_1 \wedge b, c_1\}$, we can still derive $\vdash_{A_1} [\{\mathsf{rs}\}]\ \mathsf{next}^*; \mathsf{g?}\ [\{\mathsf{gs}, \mathsf{gd}\}]$, then $\vdash_{A_1} [\{\mathsf{gs}, \mathsf{gd}\}]\ \mathsf{next}\ [\{\mathsf{gd}, \mathsf{yd}\}]$, and finally $\vdash_{A_1} [\{\mathsf{gd}, \mathsf{yd}\}]\ \neg\mathsf{d?}\ [\varnothing]$, which can be composed together to conclude $\vdash_{A_1} [\{\mathsf{rs}\}]\ \lfloor\overline{\psi}\rceil\ [\varnothing]$. In fact, in A_1 we just have $[\![\lfloor\overline{\psi}\rceil]\!]^\sharp \alpha_1(\{\mathsf{rs}\}) = \bot$.

Original contribution. We set up a theoretical framework for the systematic reduction of model checking of temporal logics to program verification, thereby enabling the re-use of abstract interpretation techniques and verifiers either directly or with minimal effort. This framework consists of:

- MOKA, a meta-programming language in which different temporal logics can be encoded so that the MOKA code for a formula φ computes exactly the counterexamples to φ;
- a systematic technique for lifting abstractions over state properties to abstractions suitable for the static analysis of MOKA programs;
- an extension of the LCL program logic to handle least fixpoints, introducing a form of "false alarm-guided" abstraction refinement loop in the analysis of MOKA programs.

Synopsis. In § 2 we provide some basics about abstract interpretation and the (fragments) of temporal logics considered in the paper. In § 3 we introduce the language MOKA, and then prove in § 4 the key results that relate formula satisfaction with program execution. In § 5 we define a general technique for deriving abstract domains for the static analysis of MOKA programs. In § 6 we showcase how local completeness logic reasoning can be exploited in our framework. In § 7 we discuss related work. Finally, in § 8 we draw some conclusions and sketch future avenues of research. Full proofs of our results and additional material are available in the extended version of this paper [1].

2 Background

A complete lattice is a poset $(L, \leq_L)$ where every subset $X \subseteq L$ has both least upper bound (lub) and greatest lower bound (glb), denoted by $\bigvee_L X$ and $\bigwedge_L X$, respectively, with $\bot_L \triangleq \bigvee_L \varnothing$ and $\top_L \triangleq \bigwedge_L \varnothing$. When no ambiguities can arise, a lattice will be denoted as L and subscripts will be omitted.

Given two complete lattices L_1 and L_2, a function $f : L_1 \to L_2$, is monotone if $x \leq_1 y$ implies $f(x) \leq_2 f(y)$, and additive (resp., co-additive) if it preserves arbitrary lub (resp., glb). Any monotone function $f : L \to L$ on a complete lattice has both a least and a greatest fixpoint, denoted by $\mathrm{lfp}(f)$ and $\mathrm{gfp}(f)$, respectively. The set of functions $f : S \to L$ from a set S to a complete lattice L, denoted L^S, forms a complete lattice when endowed with the pointwise order s.t. $f \leq g$ if for all $s \in S$, $f(s) \leq_L g(s)$. If L_1, L_2 are complete lattices, then $L_1 \times L_2$ is their product lattice endowed with the componentwise order s.t. $(x_1, x_2) \leq (y_1, y_2)$ if $x_1 \leq_1 y_1$ and $x_2 \leq_2 y_2$. We write L^n for the n-ary product of L with itself and $L^+ \triangleq \{\bot, \top\} \cup \bigcup_{n \geq 1} L^n$ for the complete lattice of non-empty finite sequences in L, ordered by $x_1 \dots x_n \leq y_1 \dots y_m$ if $n = m$ and $x_i \leq_L y_i$ for all $i \in [1, n]$, with top $\top$ and bottom $\bot$.

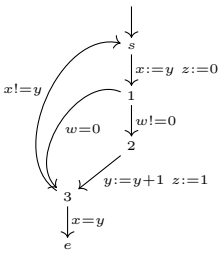
2.1 Abstract Interpretation

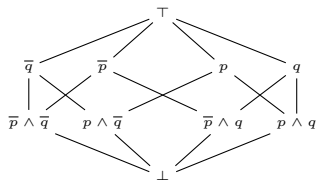
Let us recall the basics of abstract interpretation [17] (see [16] for a thorough account).

Given two complete lattices C and A, called the *concrete* and the *abstract* domain, respectively, a *Galois connection (GC)* $\langle \alpha, \gamma \rangle : C \rightleftarrows A$ is a pair of functions $\alpha : C \to A$ and $\gamma : A \to C$ s.t. $\alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$ for any $c \in C$ and $a \in A$.

The function α is referred to as abstraction map and turns out to be additive, while γ is the concretization map which is always co-additive. Intuitively, any abstract element $a \in A$ such that $c \leq \gamma(a)$ is a sound over-approximation for the concrete value c, while the abstraction $\alpha(c)$ is the most precise over-approximation of c in the abstract domain A, i.e., $\alpha(c) = \bigwedge_C \{a \mid c \leq_C \gamma(a)\}$ holds. The notation $A_{\alpha,\gamma}$ denotes an abstract domain endowed with its underlying GC, and we will omit subscripts when α and γ are clear from the context.

```
1  %
2    s  do {
3       x := y; z := 0;  1
4       if (w != 0) then {
5       2   y := y + 1;
6           z := 1; }  3
7       } while (x != y)  e
```





(a) Program c. **(b)** Control flow graph of c. **(c)** Predicate abstraction $\mathbb{P}$.

Figure 2 Program from [4, Figure 1], with control flow graph and predicate abstraction domain.

▶ **Example 2.1** (Image adjunction)**.** Given any function $f : X \to Y$, let $f^{>}$ and $f^{<}$ denote the direct and inverse image of f, respectively. The pair $\langle f^{>}, f^{<}\rangle : \mathcal{P}(X) \to \mathcal{P}(Y)$ is a GC that we refer to as the *image adjunction* (this is an instance of [24, Exercise 7.18]). ⌟

The class of abstract domains on C, denoted by $\mathrm{Abs}(C) \triangleq \{A_{\alpha,\gamma} \mid \langle \alpha, \gamma\rangle : C \rightleftarrows A\}$, can be preordered by the domain refinement relation: $A' \sqsubseteq A$ when $\gamma_A(A) \subseteq \gamma_{A'}(A')$.

▶ **Example 2.2** (Control flow graphs and predicate abstraction)**.** Any program can be represented by its control flow graph (CFG). Let Var be a set of variables valued in $\mathbb{V}$ and denote by $\mathit{Env} = \mathbb{V}^{\mathit{Var}}$ the set of environments. A CFG is a graph (N, E, s, e), where N is a finite set of nodes, representing program points, $s, e \in N$ are the start and end nodes, respectively, and $E \subseteq N \times F \times N$ is a set of edges, labelled over a set of transfer (additive) functions $F \subseteq \mathcal{P}(\mathit{Env})^{\mathcal{P}(\mathit{Env})}$. For example, the program c in Figure 2a is decorated with program points $n \in N = \{s, 1, 2, 3, e\}$, has variables $\mathit{Var} = \{x, y, z, w\}$, and values ranging in the finite domain $\mathbb{V} = \mathbb{Z}_k$ of integers modulo a given $k > 0$. The CFG of c is depicted in Figure 2b.

Predicate abstraction allows to approximate program invariants [32]. Given a set of predicates $\mathit{Pred} \subseteq \mathcal{P}(\mathit{Env})$ (where any $p \in \mathit{Pred}$ has a representation $p = \{\rho \in \mathit{Env} \mid \rho \models p\}$), the predicate abstraction domain $\mathbb{P}$ is defined by adding to Pred the complement predicates $\overline{\mathit{Pred}} \triangleq \{\overline{p} \mid p \in \mathit{Pred}\}$, and then by closing $\mathit{Pred} \cup \overline{\mathit{Pred}}$ under logical conjunction. We define a function $\pi : \mathcal{P}(\mathit{Env}) \to \mathbb{P}$ that associates to each set of environments the strongest predicate it satisfies. For example, given the two predicates $p \triangleq (z = 0)$ and $q \triangleq (x = y)$, the predicate abstraction domain $\mathbb{P}$ induced by the set $\mathit{Pred} = \{p, q\}$ is depicted in Figure 2c. Correspondingly, the product abstraction $\mathbb{P}^N$ allows us to represent the abstract state of the program as a function that associates to each program point n the strongest predicate in $\mathbb{P}$ that holds at n. Hence, for instance, the set of all possible initial states of c, which is $\{(s, xyzw) \mid x, y, z, w \in \mathbb{V}\}$, is represented by the function $(s \mapsto \top, 1 \mapsto \bot, 2 \mapsto \bot, 3 \mapsto \bot, e \mapsto \bot)$, often written $(s \mapsto \top)$, omitting the program points which are mapped to $\bot$. ⌟

An abstract interpreter computes in the underlying abstract domain through correct (and effective) abstract approximations of concrete functions. Given $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ and a function $f : C \to C$, an abstract function $f^\sharp : A \to A$ is a correct approximation of f if $\alpha \circ f \leq f^\sharp \circ \alpha$, and it is a *complete* approximation of f when $\alpha \circ f = f^\sharp \circ \alpha$. The *best correct approximation* (bca) of f in A, is defined as $f^A \triangleq \alpha \circ f \circ \gamma : A \to A$, and turns out to be the most precise correct abstraction, i.e., $f^A \leq f^\sharp$ for any other correct approximation $f^\sharp$ of f. When $f^\sharp$ is complete, then $f^\sharp = f^A$, thus making completeness an abstract domain property defined by the equation $\alpha \circ f = \alpha \circ f \circ \gamma \circ \alpha$. In program analysis, abstract domains are commonly

endowed with correct but incomplete abstract transfer functions. When completeness holds, the abstract interpreter is as precise as possible for the given abstract domain and cannot raise false alarms when verifying properties that are expressible in the abstract domain [27].

2.2 Transition Systems and Logics

Temporal logics are typically interpreted over unlabeled, finite, directed graphs, whose nodes and edges model states and transitions between them (i.e., state changes), respectively.

A *transition system* T is a tuple $(\Sigma, \mathrm{I}, \mathbf{P}, \rightarrowtriangle, \vdash)$, where Σ is a finite set of states ranged over by σ, $\mathrm{I} \subseteq \Sigma$ is the set of initial states, $\rightarrowtriangle \subseteq \Sigma \times \Sigma$ is the transition relation, $\mathbf{P}$ is a (finite) set of atomic propositions, ranged over by p, and $\vdash \subseteq \Sigma \times \mathbf{P}$ is the satisfaction relation.

We write $\sigma \rightarrowtriangle \sigma'$ instead of $(\sigma, \sigma') \in \rightarrowtriangle$ and let $\mathrm{post}(\sigma) = \{\sigma' \mid \sigma \rightarrowtriangle \sigma'\}$ denote the set of direct successors of $\sigma \in \Sigma$. As common in model checking [13], we consider systems whose transition relation is total, i.e., for all $\sigma \in \Sigma$ there exists $\sigma' \in \Sigma$ such that $\sigma \rightarrowtriangle \sigma'$.

A path is an infinite denumerable state sequence $(\sigma_i)_{i \in \mathbb{N}}$ such that $\sigma_i \rightarrowtriangle \sigma_{i+1}$ for all $i \in \mathbb{N}$.

▶ **Example 2.3** (Control flow graphs as transition systems)**.** A CFG (N, E, s, e) can be viewed as a transition system with states $\Sigma = N \times \mathit{Env}$ and transition relation defined by $(n, \rho) \rightarrowtriangle (n', \rho')$ if there is an edge $(n, f, n') \in E$ such that $\rho' \in f(\{\rho\})$. Additionally, in order to make the transition relation total we add self-loops to all the states (e, ρ) involving the end node e. For instance, by using the shorthand $(n, xyzw)$ for the states in Example 2.2, the transition $(s, 0111) \rightarrowtriangle (1, 1101)$ is induced by the edge $(s, x := y\ z := 0, 1)$ in Figure 2b, while $(3, 1111) \rightarrowtriangle (e, 1111)$ by $(3, x = y, e)$ in Figure 2b.

Let us point out that the predicate abstraction in Example 2.2 naturally lifts to the power-set of states with codomain $A = \mathbb{P}^N$, with the abstraction map $\alpha(X)(n) \triangleq \pi(\{\rho \mid (n, \rho) \in X\})$ for $X \in \mathcal{P}(\Sigma) = \mathcal{P}(N \times \mathit{Env})$. Given $\sigma^\sharp \in A$, we define $\mathit{supp}(\sigma^\sharp) \triangleq \{n \in N \mid \sigma^\sharp(n) \neq \bot\}$. ⌟

ACTL. ACTL is the fragment of CTL whose temporal formulae are universally quantified over all paths leaving the current state. Thus, given a set of atomic propositions $p \in \mathbf{P}$:

$$\mathrm{ACTL} \ni \varphi \ ::= \ p \mid \neg p \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \mathsf{AX}\ \varphi_1 \mid \mathsf{AF}\ \varphi_1 \mid \mathsf{AG}\ \varphi_1 \mid \varphi_1\ \mathsf{AU}\ \varphi_2$$

▶ **Definition 2.4** (ACTL semantics)**.** *Given a transition system $T = (\Sigma, \mathrm{I}, \mathbf{P}, \rightarrowtriangle, \vdash)$, the semantics $[\![\varphi]\!] \subseteq \Sigma$ of* ACTL *formulae over T is as follows:*

$$\begin{aligned}
[\![p]\!] &\triangleq \{\sigma \in \Sigma \mid \sigma \vdash p\} & [\![\neg p]\!] &\triangleq \{\sigma \in \Sigma \mid \sigma \nvdash p\} \\
[\![\varphi_1 \vee \varphi_2]\!] &\triangleq [\![\varphi_1]\!] \cup [\![\varphi_2]\!] & [\![\varphi_1 \wedge \varphi_2]\!] &\triangleq [\![\varphi_1]\!] \cap [\![\varphi_2]\!] \\
[\![\mathsf{AX}\ \varphi_1]\!] &\triangleq \{\sigma \in \Sigma \mid \forall \sigma'. \sigma \rightarrowtriangle \sigma' \Rightarrow \sigma' \in [\![\varphi_1]\!]\} \\
[\![\mathsf{AF}\ \varphi_1]\!] &\triangleq \{\sigma_0 \in \Sigma \mid \textit{for all path } (\sigma_i)_{i \in \mathbb{N}}\ \exists k \in \mathbb{N}.\ \sigma_k \in [\![\varphi_1]\!]\} \\
[\![\mathsf{AG}\ \varphi_1]\!] &\triangleq \{\sigma_0 \in \Sigma \mid \textit{for all path } (\sigma_i)_{i \in \mathbb{N}}\ \forall j \in \mathbb{N}.\ \sigma_j \in [\![\varphi_1]\!]\} \\
[\![\varphi_1\ \mathsf{AU}\ \varphi_2]\!] &\triangleq \{\sigma_0 \in \Sigma \mid \textit{for all path } (\sigma_i)_{i \in \mathbb{N}}\ \exists k \in \mathbb{N}.\ (\sigma_k \in [\![\varphi_2]\!] \wedge \forall j < k.\ \sigma_j \in [\![\varphi_1]\!])\}
\end{aligned}$$

Universal fragment of single variable μ-calculus. The modal μ-calculus is a well known extension of propositional modal logic with least and greatest fixed point operators. We will focus on its universal fragment only allowing the $\Box$ modal operator that quantifies over all transitions. Moreover, for the sake of simplicity, we restrict to the single variable fragment where, roughly speaking, nested fixpoints cannot have mutual dependencies.

Given a set of atomic propositions $p \in \mathbf{P}$, the $\mu_\Box$-calculus is defined as follows:

$$\mu_\Box \ni \varphi \ ::= \ p \mid \neg p \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \Box \varphi_1 \mid x \mid \mu x.\varphi_x \mid \nu x.\varphi_x$$

▶ **Definition 2.5** ($\mu_\Box$-calculus semantics). *Given $T = (\Sigma, \mathrm{I}, \mathbf{P}, \rightarrowtriangle, \vdash)$, and a valuation $\mathcal{V} : \mathrm{Var} \to \mathcal{P}(\Sigma)$, the semantics $[\![\varphi]\!]_\mathcal{V} \subseteq \Sigma$ of $\mu_\Box$-calculus formulae over T is as follows:*

$$\begin{aligned}
[\![p]\!]_\mathcal{V} &\triangleq \{\sigma \in \Sigma \mid \sigma \vdash p\} & [\![\neg p]\!]_\mathcal{V} &\triangleq \{\sigma \in \Sigma \mid \sigma \nvdash p\} \\
[\![\varphi_1 \vee \varphi_2]\!]_\mathcal{V} &\triangleq [\![\varphi_1]\!]_\mathcal{V} \cup [\![\varphi_2]\!]_\mathcal{V} & [\![\varphi_1 \wedge \varphi_2]\!]_\mathcal{V} &\triangleq [\![\varphi_1]\!]_\mathcal{V} \cap [\![\varphi_2]\!]_\mathcal{V} \\
[\![\Box\varphi_1]\!]_\mathcal{V} &\triangleq \{\sigma \in \Sigma \mid \forall \sigma'.\sigma \rightarrowtriangle \sigma' \Rightarrow \sigma' \in [\![\varphi_1]\!]_\mathcal{V}\} & [\![x]\!]_\mathcal{V} &\triangleq \mathcal{V}(x) \\
[\![\mu x.\varphi_x]\!]_\mathcal{V} &\triangleq \mathrm{lfp}(\lambda S.[\![\varphi_x]\!]_{\mathcal{V}[x \mapsto S]}) & [\![\nu x.\varphi_x]\!]_\mathcal{V} &\triangleq \mathrm{gfp}(\lambda S.[\![\varphi_x]\!]_{\mathcal{V}[x \mapsto S]})
\end{aligned}$$

where $\mathcal{V}[x \mapsto S]$ is the usual notation for function update.

We write $[\![\varphi]\!]$ instead of $[\![\varphi]\!]_\mathcal{V}$ when the valuation is inessential, and $\sigma \models \varphi$ when $\sigma \in [\![\varphi]\!]$.

3 The Language MOKA

We define a meta-language, called MOKA (for MOdel checking as abstract interpretation of Kleene Algebras), as a (generalised) Kleene Algebra with a set of basic expressions suited for identifying counterexamples to the validity of temporal formulae.

KAF. We rely on Kozen's Kleene Algebra with tests [34] with a Fixpoint operator [36], KAF for short. Given a set Exp of basic expressions e, KAF is defined below:

$$\mathsf{KAF} \ni \mathsf{r} ::= 1 \mid 0 \mid \mathsf{e} \mid \mathsf{r}_1;\mathsf{r}_2 \mid \mathsf{r}_1 \oplus \mathsf{r}_2 \mid \mathsf{r}_1^* \mid \mathsf{X} \mid \mu\mathsf{X}.\mathsf{r}_1$$

The term 1 represents the identity, i.e. no action, 0 represents divergence, $\mathsf{r}_1;\mathsf{r}_2$ represents sequential composition, $\mathsf{r}_1 \oplus \mathsf{r}_2$ represents non-deterministic choice, r_1^* represents the Kleene iteration of r_1, i.e. r_1 performed zero or any finite number of times, X is a variable ranging in a set Var, and $\mu\mathsf{X}.\mathsf{r}_1$ represents the least fixpoint operator with respect to variable X.

Commands are interpreted as functions over a complete lattice C. Given a semantics $(\!|\cdot|\!) : \mathsf{Exp} \to C \to C$ for basic expressions, the semantics of regular expressions $[\![\cdot]\!]_\eta : \mathsf{KAF} \to C \to C$ is inductively defined as follows, where $\eta : \mathsf{Var} \to C \to C$ is an environment:

$$\begin{aligned}
[\![1]\!]_\eta &\triangleq \lambda x.\, x & [\![0]\!]_\eta &\triangleq \lambda x.\, \bot \\
[\![\mathsf{e}]\!]_\eta &\triangleq (\!|\mathsf{e}|\!) & [\![\mathsf{r}_1;\mathsf{r}_2]\!]_\eta &\triangleq [\![\mathsf{r}_2]\!]_\eta \circ [\![\mathsf{r}_1]\!]_\eta \\
[\![\mathsf{r}_1 \oplus \mathsf{r}_2]\!]_\eta &\triangleq [\![\mathsf{r}_1]\!]_\eta \vee [\![\mathsf{r}_2]\!]_\eta & [\![\mathsf{r}_1^*]\!]_\eta &\triangleq \bigvee \left\{ [\![\mathsf{r}_1]\!]_\eta^k \mid k \in \mathbb{N} \right\} \\
[\![\mathsf{X}]\!]_\eta &\triangleq \eta(\mathsf{X}) & [\![\mu\mathsf{X}.\mathsf{r}_1]\!]_\eta &\triangleq \mathrm{lfp}(\lambda f : C \to C.[\![\mathsf{r}_1]\!]_{\eta[\mathsf{X} \mapsto f]})
\end{aligned}$$

It can be seen that the Kleene star r^* can be encoded as a fixpoint $\mu\mathsf{X}.(1 \oplus \mathsf{r};\mathsf{X})$, and, when the semantics of basic expressions $(\!|\cdot|\!)$ is additive, also as $\mathsf{r}^* = \mu\mathsf{X}.(1 \oplus \mathsf{X};\mathsf{r})$. Although redundant, the inclusion of the Kleene star in our syntax is very convenient, as it enables significant simplifications in some encodings, particularly when the full expressiveness of least fixpoint calculations is not necessary (see the remark at the end of Section 4). The same applies to the proof logic studied in Section 6: while basic LCL suffices for the KAF fragment without least fixpoint operator, its extension μLCL is otherwise needed. For closed KAF terms the environment is inessential and we write just $[\![\mathsf{r}]\!]$ instead of $[\![\mathsf{r}]\!]_\eta$.

The Language MOKA. Given a transition system $T = (\Sigma, \mathrm{I}, \mathbf{P}, \rightarrowtriangle, \vdash)$, MOKA is an instance of KAF interpreted over stacks of frames, each frame representing a computation path.

▶ **Definition 3.1** (Frame, stack). *A* frame *is a pair $\langle \sigma, \Delta \rangle \in \Sigma \times \mathcal{P}(\Sigma)$. We denote by F_Σ the set of frames. A* stack *is a finite non-empty sequence of frames, i.e. an element of F_Σ^+, denoted by $\langle \sigma, \Delta \rangle :: S$, where S is a stack or the empty sequence ε.*

A frame $\langle \sigma, \Delta \rangle$ represents a computation in T where σ is the current state and Δ is the set of traversed states, used for loop-detection. The order of the traversed states and their possible multiple occurrences are abstracted away as they are irrelevant when checking the satisfaction of a formula. Frames are stacked to deal with formulae with nested operators.

The language MOKA is defined as instance of KAF with basic expressions for extending and filtering frames, for (de)constructing stacks, and to keep track of fixed-point equations.

▶ **Definition 3.2** (MOKA language). *The language* MOKA *is an instance of* KAF *with the following basic expressions, where* p *ranges over atomic propositions:*

$$\mathsf{e} ::= \mathsf{p?} \mid \neg\mathsf{p?} \mid \mathsf{loop?} \mid \mathsf{next} \mid \mathsf{add} \mid \mathsf{reset} \mid \mathsf{push} \mid \mathsf{pop}$$

The (concrete) semantics of MOKA is given over the powerset of the set of stacks, $C = \mathcal{P}(\mathrm{F}^+_\Sigma)$, ordered by subset inclusion. Although we could focus on stacks of uniform length, we use a larger domain to simplify the notation.

The semantics of MOKA commands $[\![\mathsf{r}]\!]_\eta : \mathcal{P}(\mathrm{F}^+_\Sigma) \to \mathcal{P}(\mathrm{F}^+_\Sigma)$ follows from the general definition for KAF, once we specify the semantics of its basic expressions.

▶ **Definition 3.3** (Basic expression semantics). *Given* $T = (\Sigma, \mathrm{I}, \mathbf{P}, \rightarrowtriangle, \vdash)$*, the semantics of* MOKA *basic expressions is the additive extension of the functions below, where* $\langle \sigma, \Delta \rangle :: S \in \mathrm{F}^+_\Sigma$*:*

$$\begin{array}{ll}
(\!|\mathsf{p?}|\!)\,\{\langle \sigma, \Delta\rangle :: S\} \triangleq \{\langle \sigma, \Delta\rangle :: S \mid \sigma \vdash p\} & (\!|\neg\mathsf{p?}|\!)\,\{\langle \sigma, \Delta\rangle :: S\} \triangleq \{\langle \sigma, \Delta\rangle :: S \mid \sigma \nvdash p\} \\
(\!|\mathsf{loop?}|\!)\,\{\langle \sigma, \Delta\rangle :: S\} \triangleq \{\langle \sigma, \Delta\rangle :: S \mid \sigma \in \Delta\} & (\!|\mathsf{next}|\!)\,\{\langle \sigma, \Delta\rangle :: S\} \triangleq \{\langle \sigma', \Delta\rangle :: S \mid \sigma \rightarrowtriangle \sigma'\} \\
(\!|\mathsf{add}|\!)\,\{\langle \sigma, \Delta\rangle :: S\} \triangleq \{\langle \sigma, \Delta \cup \{\sigma\}\rangle :: S\} & (\!|\mathsf{reset}|\!)\,\{\langle \sigma, \Delta\rangle :: S\} \triangleq \{\langle \sigma, \varnothing\rangle :: S\} \\
(\!|\mathsf{push}|\!)\,\{\langle \sigma, \Delta\rangle :: S\} \triangleq \{\langle \sigma, \Delta\rangle :: \langle \sigma, \Delta\rangle :: S\} & (\!|\mathsf{pop}|\!)\,\{\langle \sigma, \Delta\rangle :: S\} \triangleq \{S \mid S \neq \varepsilon\}
\end{array}$$

The basic expressions p?, ¬p?, loop?, next, add, reset operate on the top frame. The filter p? (resp. ¬p?) checks the validity of the proposition p (resp. $\neg p$), loop? checks if the current state loops back to some state in the trace, next extends the trace by one step in all possible ways, add adds the current state to the trace and reset empties the trace. Instead, push and pop change the stack length: push extends the stack to start a new trace, while pop restores the previous trace from the stack. Given a set of states $X \subseteq \Sigma$, we write $[\![\mathsf{r}]\!]_\eta X$ as a shorthand for $[\![\mathsf{r}]\!]_\eta(\{\langle \sigma, \varnothing\rangle \mid \sigma \in X\})$.

4 Formulae as MOKA programs

We show that ACTL and $\mu_\Box$-calculus formulae φ can be transformed into MOKA programs $\lfloor \overline{\varphi} \rceil$ so that for any transition system T, the semantics of $\lfloor \overline{\varphi} \rceil$ over T consists exactly of the counterexamples to the validity of φ, that is, the states which do not satisfy φ.

▶ **Theorem 4.1** (Model checking as program verification). *Given* $T = (\Sigma, \mathrm{I}, \mathbf{P}, \rightarrowtriangle, \vdash)$*, for any ACTL or* $\mu_\Box$ *formula* φ *and set of stacks* $P \subseteq \mathrm{F}^+_\Sigma$*, it holds* $[\![\lfloor \overline{\varphi} \rceil]\!]_\eta P = \{\langle \sigma, \Delta\rangle :: S \in P \mid \sigma \not\models \varphi\}$.

It follows that $[\![\lfloor \overline{\varphi} \rceil]\!]_\eta P \subseteq P$ and that $[\![\lfloor \overline{\varphi} \rceil]\!]_\eta \{\sigma\} = \varnothing$ iff $\sigma \models \varphi$. It is also worth noting that the semantics of MOKA programs encoding formulae is a lower closure on sets of states, being monotone, reductive and idempotent (and it preserves arbitrary unions) [39, Section 3.2.3]. In this respect, it behaves analogously to the collecting semantics of Boolean tests which filter out memory states, keeping only those that satisfy the condition. As a consequence, similarly to what happens when abstracting Boolean tests, in any abstract domain A approximating sets of states the identity $\lambda a \in A.\, a$, is a correct (over-)approximation of $[\![\lfloor \overline{\varphi} \rceil]\!]_\eta$.

ACTL as MOKA. To each ACTL formula φ we assign the MOKA program $\lfloor\overline{\varphi}\rceil$ as follows:

$$\begin{aligned}
\lfloor\overline{p}\rceil &\triangleq \neg\mathsf{p}? & \lfloor\overline{\neg p}\rceil &\triangleq \mathsf{p}?\\
\lfloor\overline{\varphi_1 \wedge \varphi_2}\rceil &\triangleq \lfloor\overline{\varphi_1}\rceil \oplus \lfloor\overline{\varphi_2}\rceil & \lfloor\overline{\varphi_1 \vee \varphi_2}\rceil &\triangleq \lfloor\overline{\varphi_1}\rceil ; \lfloor\overline{\varphi_2}\rceil\\
\lfloor\overline{\mathsf{AX}\ \varphi}\rceil &\triangleq \mathsf{push}; \mathsf{next}; \lfloor\overline{\varphi}\rceil; \mathsf{pop}\\
\lfloor\overline{\mathsf{AF}\ \varphi}\rceil &\triangleq \lfloor\overline{\varphi}\rceil; \mathsf{push}; \mathsf{reset}; \big(\mathsf{add}; \mathsf{next}; \lfloor\overline{\varphi}\rceil\big)^* ; \mathsf{loop}?; \mathsf{pop}\\
\lfloor\overline{\mathsf{AG}\ \varphi}\rceil &\triangleq \mathsf{push}; \mathsf{next}^*; \lfloor\overline{\varphi}\rceil; \mathsf{pop}\\
\lfloor\overline{\varphi_1\ \mathsf{AU}\ \varphi_2}\rceil &\triangleq \lfloor\overline{\varphi_2}\rceil; \mathsf{push}; \mathsf{reset}; \big(\mathsf{add}; \mathsf{next}; \lfloor\overline{\varphi_2}\rceil\big)^* ; \big(\mathsf{loop}? \oplus \lfloor\overline{\varphi_1}\rceil\big) ; \mathsf{pop}
\end{aligned}$$

The intuition is that MOKA programs $\lfloor\overline{\varphi}\rceil$ act as (negative) filters: applied to a set of frames $\langle\sigma, \Delta\rangle{::}S$, each representing a computation that reached state σ, they filter out states where φ is satisfied, keeping only candidate counterexamples to the validity of φ. In general, when the check requires exploring the future of the current state, a new frame is started with push; if the search is successful, a closing pop recovers the starting state violating the formula. We explain the main clauses defining $\lfloor\overline{\varphi}\rceil$ (see [1, Theorem B.1] for the proof of the correctness of this transform). It turns out that $\langle\sigma, \Delta\rangle{::}S$ is a counterexample for

- AX φ if φ is violated in at least one successor state of σ (as computed by next; $\lfloor\overline{\varphi}\rceil$).
- AF φ if there is an infinite trace where φ never holds. This is encoded by saying that φ does not hold in the current state, i.e. $\lfloor\overline{\varphi}\rceil$, and after that, there is an infinite trace traversing only states (collected through the command $(\mathsf{add}; \mathsf{next}; \lfloor\overline{\varphi}\rceil)^*$) that do not satisfy φ. Since we deal with finite state systems, infinite traces can be identified with looping traces (whence the check loop?); for this to work, after the push we reset the past.
- AG φ if we can reach, by repeatedly applying next, a counterexample to φ.
- φ_1 AU φ_2 if φ_2 does not hold in the current state, i.e. $\lfloor\overline{\varphi_2}\rceil$, and progressing through states that do not satisfy φ_2 with $(\mathsf{add}; \mathsf{next}; \lfloor\overline{\varphi_2}\rceil)^*$, either we detect a maximal (looping) trace or a state where φ_1 does not hold.

▶ **Example 4.2** (Control flow graphs and ACTL encoding)**.** In our running example (see Examples 2.2 and 2.3), we check the property stating that if the program c terminates then the variable z is zero, which in the CFG means *"when the exit node e is reached, $z = 0$"* holds. This is expressed by the ACTL formula $\varphi = \mathsf{AG}\ (n = e \to z = 0)$, where we use $p \to \varphi'$ as syntactic sugar for $\neg p \vee \varphi'$. The corresponding MOKA program is: $\mathsf{m}_\varphi \triangleq \lfloor\overline{\varphi}\rceil = \lfloor\overline{\mathsf{AG}\ (n = e \to z = 0)}\rceil = \mathsf{push}; \mathsf{next}^*; \mathsf{n} = \mathsf{e}?; \mathsf{z} \neq 0?; \mathsf{pop}.$ ⌟

$\mu_\Box$-calculus as MOKA. To each $\mu_\Box$-formula φ we assign the MOKA program $\lfloor\overline{\varphi}\rceil$ as follows:

$$\begin{aligned}
\lfloor\overline{p}\rceil &\triangleq \neg\mathsf{p}? & \lfloor\overline{\neg p}\rceil &\triangleq \mathsf{p}?\\
\lfloor\overline{\varphi_1 \wedge \varphi_2}\rceil &\triangleq \lfloor\overline{\varphi_1}\rceil \oplus \lfloor\overline{\varphi_2}\rceil & \lfloor\overline{\varphi_1 \vee \varphi_2}\rceil &\triangleq \lfloor\overline{\varphi_1}\rceil ; \lfloor\overline{\varphi_2}\rceil\\
\lfloor\overline{\Box\varphi}\rceil &\triangleq \mathsf{push}; \mathsf{next}; \lfloor\overline{\varphi}\rceil; \mathsf{pop} & \lfloor\overline{x}\rceil &\triangleq \mathsf{X}\\
\lfloor\overline{\mu x.\varphi_x}\rceil &\triangleq \mathsf{push}; \mathsf{reset}; \mu\mathsf{X}.\Big(\mathsf{loop}? \oplus (\mathsf{add}; \lfloor\overline{\varphi_x}\rceil)\Big); \mathsf{pop} & \lfloor\overline{\nu x.\varphi_x}\rceil &\triangleq \mu\mathsf{X}.\lfloor\overline{\varphi_x}\rceil
\end{aligned}$$

The second component of a frame $\langle\sigma, \Delta\rangle$ is still used to identify looping computations. This intervenes in the encoding of least fixpoints $\mu x.\varphi_x$: when checking the formula, the current state first logged to the current frame ($\mathsf{add}; \lfloor\overline{\varphi_x}\rceil$ branch); a counterexample is found when we try to verify the fixpoint property in a state where the check has already been tried (filtered by loop?). The encoding of greatest fixpoints $\nu x.\varphi_x$ instead is simpler: searching for counterexamples naturally translates to a least fixpoint, which is offered natively by MOKA.

The correctness of this transform (see [1, Theorem B.2 and Corollary B.3]) leverages a small variation of the tableau construction in [53], with judgements roughly of the form $\sigma, \Delta \vdash \varphi$, meaning that the formula φ holds in a state σ assuming that all the states in Δ

have been visited while checking the current fixpoint subformula. Then, it can be shown that given a formula φ and a stack $\langle\sigma,\Delta\rangle{::}S$, if there is no successful tableau for $\sigma,\Delta\vdash\varphi$ then $[\![\lfloor\overline{\varphi}\rceil]\!]_\eta\,\{\langle\sigma,\Delta\rangle{::}S\}=\{\langle\sigma,\Delta\rangle{::}S\}$ holds, while $[\![\lfloor\overline{\varphi}\rceil]\!]_\eta\,\{\langle\sigma,\Delta\rangle{::}S\}=\varnothing$ otherwise.

▶ **Example 4.3** (Control flow graphs and $\mu_\Box$-calculus encoding)**.** Consider again Examples 2.2 and 2.3. We now check the property *"if after three steps we end up in program point* 3 *then we will loop forever, every* 4 *steps, on this program point 3."* It is known that regular properties of this kind cannot be expressed in ACTL (see, e.g., [54]). This property can be expressed in the $\mu_\Box$-calculus as $\psi\triangleq\Box^3(n=3\ \rightarrow\nu x.\,(n=3\ \wedge\ \Box^4 x))$, where $\Box^k$ is a shortand for $\Box\ldots\Box$ repeated k times. Letting r^k as a shorthand for a k times composition $\mathsf{r};\ldots;\mathsf{r}$ of a MOKA program r, the MOKA program encoding ψ is $\lfloor\overline{\psi}\rceil=(\mathsf{push};\mathsf{next})^3;(\mathsf{n}=3;\lfloor\overline{\nu x.\,(n=3\ \wedge\ \Box^4 x)}\rceil);\mathsf{pop}^3$, with $\lfloor\overline{\nu x.\,(n=3\ \wedge\ \Box^4 x)}\rceil=\mu\mathsf{X}.\,([\mathsf{n}\neq 3?\oplus((\mathsf{push};\mathsf{next})^4;\mathsf{X};\mathsf{pop}^4)])$. ⌟

It is well known that all ACTL formulae can be expressed as $\mu_\Box$-calculus formulae. For example, $\mathsf{AX}\,\varphi=\Box\varphi$, $\mathsf{AF}\,\varphi=\mu x.\,(\varphi\vee\Box x)$, and $\mathsf{AG}\,\varphi=\nu x.\,(\varphi\wedge\Box x)$. Hence one could obtain programs generating counterexamples for ACTL by encoding ACTL in the $\mu_\Box$-calculus and then generating the corresponding program. However, this in general produces programs which are (unnecessarily) more complex than those produced for ACTL formulae. For instance, the program for $\mathsf{AF}\,\varphi$ obtained through the $\mu_\Box$-calculus encoding above would be $\mathsf{push};\mathsf{reset};\mu\mathsf{X}.\Big(\mathsf{loop}?\oplus(\mathsf{add};\lfloor\overline{\varphi}\rceil;\mathsf{push};\mathsf{next};\mathsf{X};\mathsf{pop})\Big);\mathsf{pop}$.

5 Abstract Interpretation of MOKA

We show how to lift an abstraction over the states of a transition system to an abstraction over the domain of stacks. This is achieved stepwise by first considering an abstraction applied to each single stack, and then by merging classes of stacks through a suitable equivalence. The resulting stack abstraction will induce the abstract interpretation of MOKA programs.

Lifting the abstraction. Let $(\Sigma,I,\rightarrowtriangle)$ be a fixed transition system and let $\langle\alpha,\gamma\rangle:\mathcal{P}(\Sigma)\rightleftarrows A$ be an abstraction of state properties. We consider the lattice $\mathrm{F}_A\triangleq A\times A$, with componentwise order, whose elements $\langle\sigma^\sharp,\delta^\sharp\rangle$ are called *abstract frames*, and, in turn, F_A^+ whose elements $\langle\sigma^\sharp,\delta^\sharp\rangle{::}S^\sharp$ are called *abstract stacks*. As in the concrete case, we abbreviate $\langle\sigma^\sharp,\bot_A\rangle$ as $\sigma^\sharp$.

The abstraction map α on state properties can be extended to a frame abstraction, that by abusing the notation, we still denote by α, and is defined by $\alpha(\langle\sigma,\Delta\rangle)\triangleq\langle\alpha(\{\sigma\}),\alpha(\Delta)\rangle$. In turn, the abstraction is inductively defined on stacks as $\alpha(\langle\sigma,\Delta\rangle{::}S^n)\triangleq\alpha(\langle\sigma,\Delta\rangle){::}\alpha(S^n)$.

Now, a set of stacks is abstracted to a set of abstract stacks by first applying α pointwise using the image adjunction (see Example 2.1) and then joining classes of abstract stacks in a way which is parameterised by a suitable equivalence. Given an equivalence $\sim\,\subseteq L\times L$, we let $[x]_\sim$ denote the equivalence class of $x\in L$ w.r.t. $\sim$. Given $x\in L$ we let $\downarrow x\triangleq\{y\in L\mid y\leq x\}$.

▶ **Definition 5.1** (Equivalence adjunction)**.** *Given a complete lattice L, a* compatible equivalence *is an equivalence $\sim\subseteq L\times L$ such that for all $x\in L$, it holds that $[x]_\sim$ is closed by joins of non-empty subsets. Let $\mathcal{P}(L)_\sim\triangleq\{X\in\mathcal{P}(L)\mid\forall x\in X.\,[x]_\sim\cap X=\{x\}\}$, ordered as follows: $X\leq_\sim Y$ if for all $x\in X$ there is $y\in Y$ such that $x\sim y$ and $x\leq y$.*
Then, the pair $\langle\alpha_\sim,\gamma_\sim\rangle:\mathcal{P}(L)\rightleftarrows\mathcal{P}(L)_\sim$ defined, for $X\in\mathcal{P}(L)$, $Y\in\mathcal{P}(L)_\sim$, by

$$\alpha_\sim(X)\triangleq\left\{\bigvee(X\cap[x]_\sim)\mid x\in X\right\}\qquad\gamma_\sim(Y)\triangleq\bigcup\{[y]_\sim\cap\downarrow y\mid y\in Y\}$$

is a Galois connection, called equivalence adjunction.

In words, $\mathcal{P}(L)_\sim \subseteq \mathcal{P}(L)$ consists of the subsets of L containing at most one representative for each equivalence class, and $\alpha_\sim(X)$ abstracts X in the subset (in $\mathcal{P}(L)_\sim$) consisting of the least upper bounds of $\sim$-equivalent elements in X.

We can now define the stack abstraction parameterised by an equivalence on the lattice of abstract frames. Given an equivalence $\sim \subseteq X \times X$ on any set X, $\sim_+$ denotes the equivalence on X^+ induced by $\sim$ as follows: for all $x_1 \cdots x_n, y_1 \cdots y_m \in A^+$, $x_1 \dots x_n \sim_+ y_1 \dots y_m \in X^+$ if $n = m$ and $x_i \sim y_i$ for all $i \in \{1, \dots, n\}$.

▶ **Definition 5.2** (Stack abstraction). *Let $\sim$ be a compatible equivalence on the abstract domain A and let us extend it to the lattice of abstract frames F_A by $\langle \sigma_1^\sharp, \delta_1^\sharp \rangle \sim \langle \sigma_2^\sharp, \delta_2^\sharp \rangle$ if $\sigma_1^\sharp \sim \sigma_2^\sharp$. The* $\sim$-stack abstraction *is the domain $A^s_\sim \triangleq \mathcal{P}(\mathrm{F}_A^+)_\sim$ along with the Galois connection $\langle \alpha^s_\sim, \gamma^s_\sim \rangle : \mathcal{P}(\mathrm{F}_\Sigma^+) \rightleftarrows A^s_\sim$ defined by $\alpha^s_\sim \triangleq \alpha_{\sim_+} \circ \alpha^>$ and $\gamma^s_\sim \triangleq \alpha^< \circ \gamma_{\sim_+}$.*

In words, given a set X of stacks, its abstraction $\alpha^s_\sim(X)$ first applies pointwise the underlying α to each stack in X, and then joins equivalent abstract stacks. As corner cases we can have the identity relation $\sim = \mathrm{id}$, which joins abstract frames with identical first component, and $\sim = A \times A$, the trivial relation, which joins all abstract stacks into one.

An abstract interpreter for counterexamples. Given an abstraction for state properties $\langle \alpha, \gamma \rangle : \mathcal{P}(\Sigma) \rightleftarrows A$ and a compatible equivalence on A, following the standard approach in abstract interpretation [16], we consider an abstract semantics $[\![\mathsf{r}]\!]^\sharp_\eta : A^s_\sim \to A^s_\sim$, defined inductively as explained in § 3 and using as abstract semantics for basic expressions e their BCAs on $A^s_\sim$, denoted by $(\!|\mathsf{e}|\!)^{A^s_\sim}$. We write $[\![\mathsf{r}]\!]^\sharp$ instead of $[\![\mathsf{r}]\!]^\sharp_\eta$ when the abstract environment is inessential. Notably, the BCAs of basic expressions independent from the underlying system can be effectively defined, and some of them result to be complete.

▶ **Theorem 5.3** (Basic abstract operations). *Let $\sim$ be a compatible equivalence on the abstract domain A. The BCAs of the basic expressions* add, reset, push, pop *for the $\sim$-stack abstraction are as follows: for all $\langle \sigma^\sharp, \delta^\sharp \rangle :: S^\sharp \in \mathrm{F}_A^+$*

$$(\!|\mathsf{add}|\!)^{A^s_\sim}\left\{\langle \sigma^\sharp, \delta^\sharp\rangle :: S^\sharp\right\} = \left\{\langle \sigma^\sharp, \delta^\sharp \vee \sigma^\sharp\rangle :: S^\sharp\right\} \qquad (\!|\mathsf{reset}|\!)^{A^s_\sim}\left\{\langle \sigma^\sharp, \delta^\sharp\rangle :: S^\sharp\right\} = \left\{\langle \sigma^\sharp, \bot\rangle :: S^\sharp\right\}$$
$$(\!|\mathsf{push}|\!)^{A^s_\sim}\left\{\langle \sigma^\sharp, \delta^\sharp\rangle :: S^\sharp\right\} = \left\{\langle \sigma^\sharp, \delta^\sharp\rangle :: \langle \sigma^\sharp, \delta^\sharp\rangle :: S^\sharp\right\} \qquad (\!|\mathsf{pop}|\!)^{A^s_\sim}\left\{\langle \sigma^\sharp, \delta^\sharp\rangle :: S^\sharp\right\} = \left\{S^\sharp\right\}$$

and the operations reset, push, pop *are globally complete. Moreover*

$$(\!|\mathsf{p?}|\!)^{A^s_\sim}\left\{\langle \sigma^\sharp, \delta^\sharp\rangle :: S^\sharp\right\} \leq_\sim \left\{\langle \sigma^\sharp \wedge \mathsf{p?}^{A^s_\sim}, \delta^\sharp\rangle :: S^\sharp\right\}$$
$$(\!|\neg\mathsf{p?}|\!)^{A^s_\sim}\left\{\langle \sigma^\sharp, \delta^\sharp\rangle :: S^\sharp\right\} \leq_\sim \left\{\langle \sigma^\sharp \wedge \neg\mathsf{p?}^{A^s_\sim}, \delta^\sharp\rangle :: S^\sharp\right\}$$
$$(\!|\mathsf{loop?}|\!)^{A^s_\sim}\left\{\langle \sigma^\sharp, \delta^\sharp\rangle :: S^\sharp\right\} \leq_\sim \left\{\langle \sigma^\sharp, \delta^\sharp\rangle :: S^\sharp \mid \sigma^\sharp \wedge \delta^\sharp \neq \bot\right\}$$

where $\mathsf{p?}^{A^s_\sim} = \alpha([\![\mathsf{p?}]\!]\Sigma)$ is the abstraction of the set of concrete states satisfying p (and similarly for $\neg\mathsf{p?}^{A^s_\sim}$).

For loop? one can get $(\!|\mathsf{loop?}|\!)^{A^s_\sim}\left\{\langle \sigma^\sharp, \delta^\sharp\rangle :: S^\sharp\right\} = \left\{\langle \sigma^\sharp \wedge \delta^\sharp, \delta^\sharp\rangle :: S^\sharp \mid \sigma^\sharp \wedge \delta^\sharp \neq \bot\right\}$ under additional conditions on $\sim$ (cf. [1, Definition C.6]).

The soundness-by-design of this abstract semantics entails a sound program verification.

▶ **Proposition 5.4** (Program verification for satisfaction). *Given a transition system $T = (\Sigma, \mathrm{I}, \mathbf{P}, \rightarrowtriangle, \vdash)$, for all ACTL or $\mu_\Box$-calculus formulae φ, abstract domain A, and a compatible equivalence $\sim$ on A, if $[\![\lfloor\overline{\varphi}\rceil]\!]^\sharp_\eta(\alpha^s_\sim(I)) = \bot$ then, for all $\sigma \in I$, we have $\sigma \models \varphi$.*

As in the concrete case, the abstract semantics of programs encoding formulae is a lower closure. This, together with the observation that the semantics only depends on the top frame of a stack, is relevant for the concrete implementation.

▶ Remark 5.5. We recalled in § 4 that for any Boolean test b and abstract domain A, the identity function $\lambda a \in A.\, a$ is a correct approximation of the filtering semantics $[\![b]\!]$ of b. This function can be enhanced to $\lambda a \in A.\, \alpha([\![b]\!]\Sigma) \wedge_A a$, which is also a correct approximation of $[\![b]\!]$ [39, Section 4.1]. If A is a partitioning abstraction then one can show that $\lambda a \in A.\, \alpha([\![b]\!]\Sigma) \wedge_A a$ is indeed the *best* correct approximation of $[\![b]\!]$. However, this property does not hold in general, even assuming that A is a disjunctive abstraction, i.e., the additivity of γ (this is substantiated in [1, Example C.10]). ⌟

▶ **Example 5.6** (Control flow graphs and abstract frames)**.** In our running example, abstract frames are of the shape $A \times A = \mathbb{P}^N \times \mathbb{P}^N$. The abstraction of singletons is $\alpha(\{(n, \rho)\}) = (n \mapsto a)$ with $a(n') = \bot$ for all $n' \neq n$. With the aim of joining past states when they correspond to the same program point, we consider the equivalence $\sim \subseteq \mathbb{P}^N \times \mathbb{P}^N$ on abstract frames defined by $\sigma_1^\sharp \sim \sigma_2^\sharp$ when $supp(\sigma_1^\sharp) = supp(\sigma_2^\sharp)$. The property $\psi = \Box^3(n = 3 \ \rightarrow \nu x.\, (n = 3 \ \wedge \ \Box^4 x))$ from Example 4.3 holds in the system, and we can prove it with this abstraction. Let $\sigma^\sharp = (s \mapsto \top) = \alpha(\{(s, xyzw)\})$ we can compute $[\![\lfloor \overline{\psi} \rceil]\!]^\sharp_\eta(\sigma^\sharp) = \bot$, implying that the formula holds from any initial states (convergence is after a single full iteration). Instead, the abstract computation for $\varphi = \mathsf{AG}\ (n = e \rightarrow z = 0)$ from Example 4.2 yields a false positive. ⌟

6 Locally Complete Analyses

If the abstract interpretation of a MOKA program returns an alarm, i.e., $[\![\lfloor \overline{\psi} \rceil]\!]^\sharp \alpha^s_\sim(I) \neq \bot$, then any initial state in the concretisation of $[\![\lfloor \overline{\psi} \rceil]\!]^\sharp \alpha^s_\sim(I)$ is a candidate counterexample to the validity of ψ. However, due to over-approximation, we cannot distinguish spurious counterexamples from true ones. Here, we discuss how to combine under- and over-approximation for the analysis of MOKA programs $\lfloor \overline{\psi} \rceil$ to overcome this problem. In particular we leverage Local Completeness Logic (LCL) [7, 9] possibly paired with Abstract Interpretation Repair (AIR) strategies [8], to improve the analysis precision.

LCL. Most abstract domains are not globally complete for program analysis, so that the corresponding analyses may well yield false alarms. Accordingly, [7, 9] studies how completeness can be locally weakened to an analysis of interest: given a function $f : C \to C$, an abstract domain $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ is *locally complete on a value* $c \in C$, denoted by $\mathbb{C}^A_c(f)$, when $\alpha \circ f(c) = \alpha \circ f \circ \gamma \circ \alpha(c)$ (hence global completeness amounts to $\mathbb{C}^A_c(f)$ for all $c \in C$). Intuitively, the absence of false alarms in an abstract computation comes as a consequence of the local completeness of the abstract transfer functions on the traversed concrete states.

Moreover, local completeness is a convex property, and this allows to check local completeness on suitable under-approximations $u \leq c$ such that $\alpha(u) = \alpha(c)$, as $\mathbb{C}^A_u(f)$ implies $\mathbb{C}^A_c(f)$. The work [7, 9] implements this idea through LCL, an under-approximating program logic (in the style of incorrectness logic [43]), parameterised by the abstract domain A which provides an over-approximation. The LCL proof rules for KAT programs are recalled in Table 1. A provable LCL triple $\vdash_A [P]\ \mathsf{r}\ [Q]$ ensures that each state satisfying Q is reachable from some state satisfying P, and also guarantees that Q and $[\![\mathsf{r}]\!]P$ have the same abstraction in A. This means that the LCL program logic is sound w.r.t. the following notion of validity.

▶ **Definition 6.1** (Valid LCL triples)**.** *Let $P, Q \in C$ and* r *be a* KAT *program. A program triple $\vdash_A [P]\ \mathsf{r}\ [Q]$ is valid if $\mathbb{C}^A_P(\mathsf{r}) \ \wedge \ Q \leq [\![\mathsf{r}]\!]P \ \wedge \ \alpha([\![\mathsf{r}]\!]P) = \alpha(Q)$.*

The condition $\mathbb{C}^A_P(\mathsf{r})$ states that A is locally complete for $[\![\mathsf{r}]\!]$ on P, while $Q \leq [\![\mathsf{r}]\!]P$ that Q under-approximates $[\![\mathsf{r}]\!]P$ and $\alpha([\![\mathsf{r}]\!]P) = \alpha(Q)$ that $\gamma(\alpha(Q))$ over-approximates $[\![\mathsf{r}]\!]P$.

Table 1 Rules of the Local Completeness Logic [9].

$$\frac{\mathbb{C}^A_P(\mathsf{e})}{\vdash_A [P]\ \mathsf{e}\ [[\![\mathsf{e}]\!]P]}\ \text{(transfer)} \qquad \frac{P' \le P \le \gamma(\alpha(P')) \quad Q \le Q' \le \gamma(\alpha(Q)) \quad \vdash_A [P']\ \mathsf{r}\ [Q']}{\vdash_A [P]\ \mathsf{r}\ [Q]}\ \text{(relax)}$$

$$\frac{\vdash_A [P]\ \mathsf{r}_1\ [W] \quad \vdash_A [W]\ \mathsf{r}_2\ [Q]}{\vdash_A [P]\ \mathsf{r}_1;\mathsf{r}_2\ [Q]}\ \text{(seq)} \qquad \frac{\vdash_A [P]\ \mathsf{r}_1\ [Q_1] \quad \vdash_A [P]\ \mathsf{r}_2\ [Q_2]}{\vdash_A [P]\ \mathsf{r}_1 \oplus \mathsf{r}_2\ [Q_1 \vee Q_2]}\ \text{(join)}$$

$$\frac{\vdash_A [P]\ \mathsf{r}\ [W] \quad \vdash_A [P \vee W]\ \mathsf{r}^*\ [Q]}{\vdash_A [P]\ \mathsf{r}^*\ [Q]}\ \text{(rec)} \qquad \frac{\vdash_A [P]\ \mathsf{r}\ [Q] \quad Q \le \gamma(\alpha(P))}{\vdash_A [P]\ \mathsf{r}^*\ [P \vee Q]}\ \text{(iterate)}$$

Table 2 Novel rules for μLCL.

$$\frac{}{\vdash_A [P]\ \mu^0\mathsf{X}.\mathsf{r}\ [\bot]}\ (\mu^0) \qquad \frac{\vdash_A [P]\ \mathsf{r}[\mu^n\mathsf{X}.\mathsf{r}/\mathsf{X}]\ [Q]}{\vdash_A [P]\ \mu^{n+1}\mathsf{X}.\mathsf{r}\ [Q]}\ (\mu^+) \qquad \frac{[\![\mu\mathsf{X}.\mathsf{r}]\!]^\sharp\alpha(P) \le_A \alpha(Q) \quad \vdash_A [P]\ \mu^n\mathsf{X}.\mathsf{r}\ [Q]}{\vdash_A [P]\ \mu\mathsf{X}.\mathsf{r}\ [Q]}\ \text{(fix)}$$

LCL for MOKA. The LCL proof system has been defined for KAT programs only, hence it is enough for MOKA programs induced by ACTL formulae. For analysing general KAF and thus MOKA programs, we extend it to μLCL including the inference rules in Table 2, where $\mathsf{r}[\mathsf{s}/\mathsf{X}]$ denotes the capture-avoiding substitution of the free occurrences of X for s in r and the term $\mu^n\mathsf{X}.\mathsf{r}$ represents the n-th fixpoint approximant with the expected semantics ($[\![\mu^0\mathsf{X}.\mathsf{r}]\!]_\eta \triangleq \lambda x.\bot$ and $[\![\mu^{n+1}\mathsf{X}.\mathsf{r}]\!]_\eta \triangleq [\![\mathsf{r}]\!]_{\eta[\mathsf{X}\mapsto[\![\mu^n\mathsf{X}.\mathsf{r}]\!]_\eta]}$). The syntax $\mu^n\mathsf{X}.\mathsf{r}$ is used only in μLCL derivations. Intuitively, since $\mu^0\mathsf{X}.\mathsf{r}$ behaves as $\mathsf{0}$, the unique valid judgement is $\vdash_A [P]\ \mu^0\mathsf{X}.\mathsf{r}\ [\bot]$. The rule (μ^+) serves just to unfold $\mu^{n+1}\mathsf{X}.\mathsf{r}$ as many times as needed. The key rule is (fix) which can be used whenever the n-th approximant provides enough information: in case the premise holds, by local completeness we will have $[\![\mu^n\mathsf{X}.\mathsf{r}]\!]^\sharp\alpha(P) = \alpha(Q)$. Since it is always the case that $[\![\mu^n\mathsf{X}.\mathsf{r}]\!]^\sharp\alpha(P) \le_A [\![\mu\mathsf{X}.\mathsf{r}]\!]^\sharp\alpha(P)$, if the side condition $[\![\mu\mathsf{X}.\mathsf{r}]\!]^\sharp\alpha(P) \le_A \alpha(Q)$ holds, then local completeness for the fixpoint term $\mu\mathsf{X}.\mathsf{r}$ can be inferred. These novel rules are sound, in the sense of preserving validity (Definition 6.1).

▶ Remark 6.2. When a least fixpoint appears in $\lfloor\overline{\varphi}\rceil$, we can exploit the following heuristic for choosing the value of n: if the abstract domain satisfies the Ascending Chain Condition (ACC) then, for every $P \in C$ there is $n_P \in \mathbb{N}$ such that $[\![\mu^{n_P}\mathsf{X}.\mathsf{r}]\!]^\sharp\alpha(P) = [\![\mu\mathsf{X}.\mathsf{r}]\!]^\sharp\alpha(P)$. Then, by taking any $n \ge n_P$ such that $\vdash_A [P]\ \mu^n\mathsf{X}.\mathsf{r}\ [Q]$ is provable, the condition $[\![\mu\mathsf{X}.\mathsf{r}]\!]^\sharp\alpha(P) \le_A \alpha(Q)$ is readily satisfied, so that $\vdash_A [P]\ \mu\mathsf{X}.\mathsf{r}\ [Q]$ is valid. More precisely, the rule below is sound:

$$\frac{\vdash_A [P]\ \mu^n\mathsf{X}.\mathsf{r}\ [Q] \qquad [\![\mu^n\mathsf{X}.\mathsf{r}]\!]^\sharp\alpha(P) = [\![\mu\mathsf{X}.\mathsf{r}]\!]^\sharp\alpha(P)}{\vdash_A [P]\ \mu\mathsf{X}.\mathsf{r}\ [Q]}\ \text{(afix)}$$ ⌟

The following result relating μLCL proofs with validity of formulae follows as an easy consequence of [9, Corollary 5.6] and Theorem 4.1.

▶ **Corollary 6.3** (Precision). *Let $T = (\Sigma, \mathrm{I}, \mathbf{P}, \rightarrowtriangle, \vdash)$ be a transition system. For all ACTL or $\mu_\Box$-calculus formulae φ, abstract domain A and compatible equivalence $\sim$ on A, if the triple $\vdash_{A^s_\sim} [I]\ \lfloor\overline{\varphi}\rceil\ [Q]$ is derivable, then $Q \subseteq I$ and*

- *$Q = \varnothing$ if and only if for all $\sigma \in I$ we have $\sigma \models \varphi$;*
- *if $Q \neq \varnothing$ then for all $\sigma \in Q$ we have $\sigma \not\models \varphi$.*

▶ **Example 6.4** (Control flow graphs and LCL derivations)**.** As pointed out in Example 5.6 the computation of the program m_φ, which encodes in MOKA the property $\varphi = \mathsf{AG}\ (n = e \to z = 0)$ (see Example 4.2), yields a false alarm when we use the abstract domain A depicted in Figure 2c. Consider as set of initial states $I_0 = \{\langle(s, 0100), \varnothing\rangle, \langle(s, 0011), \varnothing\rangle\}$, whose abstraction covers the set of possible initial states $\{\langle(s, xyzw), \varnothing\rangle \mid x, y, z, w \in \mathbb{Z}_k\}$, because $\alpha^s_\sim(I_0) = (s \mapsto \top)$. Since the property holds, trying to derive a triple $\vdash_{A^s_\sim} [I_0]\ \mathsf{m}_\varphi\ [\varnothing]$ leads to the failure of some local completeness assumption, which can drive the refinement of A. The imprecision is found in the fourth iteration of the next operator, in which the set $\{(3, 1100), (3, 0111)\}$ is abstracted to $(3, xyzw)$, losing the relation between p and q being either both valid or both invalid. Several domain repairs are possible. Following [8] we can repair the domain by adding the abstract point $q \to p$ (a more abstract repair than adding the element $q \leftrightarrow p$, as proposed in [4]). Here we can also exploit a different route, by refining the equivalence so that $\sigma^\sharp_1 \sim \sigma^\sharp_2$ when $supp(\sigma^\sharp_1) = supp(\sigma^\sharp_2) \neq 3$. This intuitively corresponds to abstract separately the states at program point $n = 3$. In both cases, it holds $[\![\lfloor\overline{\varphi}\rceil]\!]^\sharp \alpha'^s_\sim(I_0) = \bot$ in the refined abstract domain A'. ⌟

▶ **Example 6.5** (Traffic light example, LCL derivation and repair, μ-calculus version)**.** Consider the toy traffic light example from Figure 1a and the property $\psi = \mathsf{AG}\ (\mathsf{g} \to \mathsf{AX}\ \mathsf{d})$, briefly discussed in the introduction. We give some additional details in the light of the results presented. We first translate the formula in $\mu_\Box$-calculus as $\psi_\mu = \nu x.((\mathsf{g} \to \Box\mathsf{d}) \wedge \Box x)$ whose encoding is $\lfloor\overline{\psi_\mu}\rceil = \mu\mathsf{X}.(\mathsf{r}_1 \oplus \mathsf{push}; \mathsf{next}; \mathsf{X}; \mathsf{pop}) \triangleq \mu\mathsf{X}.\mathsf{r}_x$, where $\mathsf{r}_1 \triangleq \mathsf{g}?; \mathsf{push}; \mathsf{next}; \neg\mathsf{d}?; \mathsf{pop}$. Then we try to derive the triple $\vdash_{A^s_\sim} [\{\mathsf{rs}\}]\ \mu\mathsf{X}.\ \mathsf{r}_x[\varnothing]$ in the abstract domain A (see Figure 1c) by applying the rule (afix) with $n = 2$ (the number of iterations needed for convergence of the abstract fixpoint in A). By definition, $\mu^0\mathsf{X}.\mathsf{r}_x = \mathsf{0}$; then the first approximant is $\mu^1\mathsf{X}.\mathsf{r}_x = \mathsf{r}_1 \oplus \mathsf{push}; \mathsf{next}; \mathsf{0}; \mathsf{pop}$, but we can omit the branch that contains $\mathsf{0}$, so $\mu^1\mathsf{X}.\mathsf{r}_x = \mathsf{r}_1$; and the second approximant is $\mathsf{r}_2 \triangleq \mu^2\mathsf{X}.\mathsf{r}_x = \mathsf{r}_1 \oplus \mathsf{push}; \mathsf{next}; \mathsf{r}_1; \mathsf{pop}$. The attempt to derive the triple $\vdash_{A^s_\sim} [\{\mathsf{rs}\}]\ \mathsf{r}_2\ [\varnothing]$ is sketched below, where the mid row gives an indication of the triples labelling the leaves of the derivation, reporting for each basic command involved, the pre-condition and post-condition of the triple, while the top row reports their abstractions.

$$
\begin{array}{cccccccccccc}
 & a & \bot & \bot & \bot & \bot & \bot & a & a & a \vee c & c & c \\
 & [\mathsf{rs}] & [\varnothing] & [\varnothing] & [\varnothing] & [\varnothing] & [\varnothing] & [\mathsf{rs}] & \left[\begin{smallmatrix}\mathsf{rs}\\ \cdots\end{smallmatrix}\right] & \left[\begin{smallmatrix}\mathsf{rs} & \mathsf{gs}\\ \cdots & \cdots\end{smallmatrix}\right] & \left[\begin{smallmatrix}\mathsf{gs}\\ \cdots\end{smallmatrix}\right] & \left[\begin{smallmatrix}\mathsf{gs}\\ \cdots\end{smallmatrix}\right] & \mathbf{!} \\
\vdash_{A^s_\sim} & [\mathsf{rs}] & \mathsf{g}?\,; & \mathsf{push}\,; & \mathsf{next}\,; & \neg\mathsf{d}?\,; & \mathsf{pop} & \oplus & \mathsf{push}\,; & \mathsf{next}\,; & \mathsf{g}?\,; & \mathsf{push}\,; \ \mathsf{next}\,; \ \neg\mathsf{d}?\,; \ \mathsf{pop}\,; \ \mathsf{pop} \ [\varnothing]
\end{array}
$$

While we can derive $\vdash_{A^s_\sim} [\{\mathsf{rs}\}]\ \mathsf{r}_1[\varnothing]$, the source of local incompleteness is found in the second branch, at the **!** position, where the proof obligation for $\mathbb{C}^{A^s_\sim}_{\{\mathsf{gs}\}}(\mathsf{next})$ fails, since $\alpha^s_\sim \circ [\![\mathsf{next}]\!](\{\mathsf{gs}\}) = \alpha^s_\sim(\{\mathsf{gd}, \mathsf{yd}\}) = b \wedge c$, while $\alpha^s_\sim \circ [\![\mathsf{next}]\!] \circ \gamma^s_\sim \circ \alpha^s_\sim(\{\mathsf{gs}\}) = \alpha^s_\sim \circ [\![\mathsf{next}]\!]\ \{\mathsf{gs}, \mathsf{gd}, \mathsf{gs}, \mathsf{ys}\} = \alpha^s_\sim(\{\mathsf{gs}, \mathsf{gd}, \mathsf{gs}, \mathsf{ys}, \mathsf{rs}\}) = a \vee c$. We can thus repair the abstract domain A. Following the procedure in [8], we add the new abstract point c_1:

$$c_1 \triangleq \bigvee \{T \subseteq \Sigma \mid T \subseteq \gamma \circ \alpha\,\{\mathsf{gs}\}\,,\ [\![\mathsf{next}]\!]T \subseteq [\![\mathsf{next}]\!](\{\mathsf{gs}\})\} = \{\mathsf{gs}, \mathsf{gd}\}$$

The repaired domain is $A_1 \triangleq A \cup \{c_1, b \wedge c_1\}$, where $b \wedge c_1$ is also added because the domain must be closed under meets. Now, in A_1, the proof obligation for $\mathbb{C}^{A^s_{1\sim}}_{\{\mathsf{gs}\}}(\mathsf{next})$ holds true: $[\![\mathsf{next}]\!](\{\mathsf{gs}\}) = \alpha_1{}^s_\sim \circ [\![\mathsf{next}]\!] \circ \gamma \circ \alpha_1{}^s_\sim(\{\mathsf{gs}\}) = b \wedge c$. In fact, in A_1 we just have $[\![\lfloor\overline{\psi}\rceil]\!]^\sharp \alpha_1{}^s_\sim(\{\mathsf{rs}\}) = \bot$ (we omit the details for the first branch r_1 that are as above):

$$
\begin{array}{cccccccccccc}
 & a & \bot & a & a & a \vee c & c_1 & c_1 & b \wedge c & \bot & \bot & \bot \\
 & [\mathsf{rs}] & [\varnothing] & [\mathsf{rs}] & \left[\begin{smallmatrix}\mathsf{rs}\\ \cdots\end{smallmatrix}\right] & \left[\begin{smallmatrix}\mathsf{rs} & \mathsf{gs}\\ \cdots & \cdots\end{smallmatrix}\right] & \left[\begin{smallmatrix}\mathsf{gs}\\ \cdots\end{smallmatrix}\right] & \left[\begin{smallmatrix}\mathsf{gs}\\ \cdots\end{smallmatrix}\right] & \left[\begin{smallmatrix}\mathsf{gd} & \mathsf{yd}\\ \cdots & \cdots\end{smallmatrix}\right] & [\varnothing] & [\varnothing] & [\varnothing] \\
\vdash_{A^s_{1\sim}} & [\mathsf{rs}] & \mathsf{r}_1 & \oplus & \mathsf{push}\,; & \mathsf{next}\,; & \mathsf{g}?\,; & \mathsf{push}\,; & \mathsf{next}\,; & \neg\mathsf{d}?\,; & \mathsf{pop}\,; & \mathsf{pop}\ [\varnothing]
\end{array}
$$

⌟

7 Related Work

Abstract interpretation and model checking have been applied to and combined with each other in several ways [2, 4, 5, 10–12, 14, 19–22, 25, 26, 28–30, 33, 35, 37, 38, 44–47, 49, 50] (see the survey [23] for further references). Abstract model checking broadly refers to checking a (temporal) specification in an abstract rather than a concrete model. On the one hand, in state partition-based approaches, the abstract model is itself a transition system, possibly induced by a behavioural state equivalence such as (bi)simulation, so that the verification method for the abstract model remains unaltered, e.g., [10–12, 14, 37]. On the other hand, if the abstract model derives from any, possibly nonpartitioning, state abstraction then the verification on this abstract model relies on abstract interpretation, e.g., [5, 21, 25, 38, 45, 46]. Hybrid abstraction techniques, in between these two approaches, have also been studied, e.g., predicate abstraction [28, 33], and the mixed transition systems with several universal and/or existential abstract transition relations in [3, 22]. Moreover, several works investigated the role of complete/exact/strongly preserving abstractions in model checking, e.g., [20, 26, 37, 44], and how to refine abstract models, e.g., [11, 12, 19, 21, 25, 29, 30, 45, 47, 50]. Let us also mention that [2] develops a theory of approximation for systems of fixpoint equations in the style of abstract interpretation, showing that up-to techniques can be interpreted as abstractions.

Our refinement technique somehow resembles CounterExample-Guided Abstraction Refinement (CEGAR) [12], a popular method for automatising the abstraction generation in model checking. CEGAR deals with state partition abstractions, thus merging sets of equivalent states: one starts from a rough abstraction which is iteratively refined on the basis of spurious counterexample traces arising due to over-approximation. The approach is sound for safety properties, i.e., no false positives can be found, and complete for a significant fragment of ACTL*. It turns out that state partition abstractions are a specific instance in our approach. The attempt of proving the absence of counterexamples in LCL using an initial coarse abstraction will yield a computation by some means similar to CEGAR: failing LCL proof obligations lead to abstraction refinements which can be more general than partitions.

The general idea that model checking can be expressed as static analysis has been first investigated in [40]. This work shows that model checking of ACTL can be reduced to a logic-based static analysis formulated within the flow logic approach [42], which is then computed through a solver for the alternation-free least fixed point logic designed in [41]. This reduction technique has been later extended to the μ-calculus in [55]. One major goal of [40] was to show the close relationship and interplay between model checking and static analysis, coupled with early work in [51] and, later, in [48, 49, 52], proving that static program analysis can be reduced to model checking of modal formulae. Let us remark that the reduction of [40, 55] to a flow logic-based static analysis is given for concrete model checking only and does not encompass the chance of dealing with abstract model checking and related abstraction refinement techniques such as CEGAR, that can be instead achieved by-design in our approach.

8 Conclusion and future work

We have introduced a framework where model checking of temporal formulae in ACTL or in the universal fragment of the modal μ-calculus can be reduced to program verification, paving the way to reuse the full range of abstract interpretation techniques. Formulae are mapped to programs of the MOKA language, that are then analysed through a sound-by-construction abstract interpretation. This exposes all the possible counterexamples, although false alarms can arise. We show how false alarms can be removed by inspecting the derivability of suitable

judgements in LCL, a program logic exploiting under- and over-approximations leveraging the notion of locally complete abstract interpretation. We expect that our approach, relying on KAF, naturally applies also to logics including operators based on regular expressions (see, e.g., [6, 31]). A first candidate is Propositional Dynamic Logic (PDL) [31], a modal logic closely connected to KAT. Its distinctive feature is a modal operator $[\mathsf{r}]\varphi$ where r is a KAT expression, which is satisfied by a state σ when all the computations of r starting from σ end up in a state satisfying φ. For instance, the property expressed in the $\mu_{\Box}$-calculus in Example 4.3 can be equivalently written in PDL as $[\mathsf{next}]^3(n = 3 \to ([\mathsf{next}]^4)^*(n = 3))$, where, since we work in an unlabelled setting, "next" stands for the only action in the system. Indeed, PDL smoothly fits in our setting (we refer the interested reader to [1, Appendix E]).

Future Work. A number of interesting avenues of future research remain open. Our results are limited to universal fragments of temporal logics and finite state domains. A dual theory can be easily developed for existential fragments, focusing on the generation of witnesses rather than counterexamples. It would be interesting to combine the two approaches for dealing with universal and existential operators at the same time. Some ideas could come from [22] that, for solving the problem, works with two different abstract transition relations. The restriction to finite state domains is due to the fact that our encoding of logical formulae into MOKA programs relies on the loop? operator for detecting infinite traces which are identified with looping traces. Further work could overcome this restriction by considering an encoding that captures non-looping infinite traces in the concrete domain and by exploiting ACC domains for the abstraction.

The use of LCL allows us to track the presence of false alarms back to the failure of local completeness proof obligations, which can be resolved by refining the abstract domain. This can be done at different levels: either refining the abstraction over states or refining the equivalence on abstract frames. A proper theory of refinements, possibly identifying optimal ways of patching the domain, is a matter of future investigations.

We point out that providing a general bound for the complexity of the abstract model checking procedure is not straightforward, as it crucially depends on the choice of the abstract domain. Different domains may induce significantly different behaviors, especially for non-partitioning abstractions or when local completeness and domain refinement techniques are applied. A precise complexity analysis tailored to specific domains is an interesting subject for future work.

Concerning the automatisation, the abstract interpreter could be easily implementable leveraging the standard toolset of abstract interpretation. The abstract interpreter should be instrumented to report, when the result is not $\bot$, an abstract counterexample trace to check whether the counterexample is a false or true alarm. Making refinements effective requires working in a class of domains where local refinements are representable, e.g., by predicate abstractions or state partition abstractions. Developing a theory of refinements within specific subclasses of domains is a direction of future work.

References

1 Paolo Baldan, Roberto Bruni, Francesco Ranzato, and Diletta Rigo. Model checking as program verification by abstract interpretation (extended version). *CoRR*, abs/2506.05525, 2025. URL: `https://arxiv.org/abs/2506.05525`.

2 Paolo Baldan, Barbara König, and Tommaso Padoan. Abstraction, up-to techniques and games for systems of fixpoint equations. In *Proceedings of CONCUR 2020*, volume 171 of *LIPIcs*, pages 25:1–25:20, 2020. `doi:10.4230/LIPICS.CONCUR.2020.25`.

3 Thomas Ball, Orna Kupferman, and Greta Yorsh. Abstraction for falsification. In *Proceedings of CAV 2005*, volume 3576 of *LNCS*, pages 67–81. Springer, 2005. doi:10.1007/11513988_8.

4 Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking C programs. In *Proceedings of TACAS 2001*, volume 2031 of *LNCS*, pages 268–283. Springer, 2001. doi:10.1007/3-540-45319-9_19.

5 Gourinath Banda and John P. Gallagher. Constraint-based abstract semantics for temporal logic: A direct approach to design and implementation. In *Proceedings of LPAR 2016*, volume 6355 of *LNCS*, pages 27–45. Springer, 2010. doi:10.1007/978-3-642-17511-4_3.

6 Ilan Beer, Shoham Ben-David, and Avner Landver. On-the-fly model checking of RCTL formulas. In *Proceedings of CAV 1998*, volume 1427 of *LNCS*, pages 184–194. Springer, 1998. doi:10.1007/BFB0028744.

7 Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. A logic for locally complete abstract interpretations. In *Proceedings of LICS 2021*. IEEE, 2021. doi:10.1109/LICS52264.2021.9470608.

8 Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. Abstract interpretation repair. In *Proceedings of PLDI 2022*, pages 426–441. ACM, 2022. doi:10.1145/3519939.3523453.

9 Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. A correctness and incorrectness program logic. *Journal of the ACM*, 70(2):15:1–15:45, 2023. doi:10.1145/3582267.

10 Doron Bustan and Orna Grumberg. Simulation-based minimization. *ACM Transactions on Computational Logic*, 4(2):181–206, 2003. doi:10.1145/635499.635502.

11 Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of CAV 2000*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000. doi:10.1007/10722167_15.

12 Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003. doi:10.1145/876638.876643.

13 Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmut Veith. *Model Checking, 2nd Edition*. MIT Press, 2018. URL: https://mitpress.mit.edu/books/model-checking-second-edition.

14 Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming, Languages and Systems*, 16(5):1512–1542, 1994. doi:10.1145/186025.186051.

15 Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model checking and the state explosion problem. In *Tools for Practical Software Verification, LASER, International Summer School 2011*, volume 7682 of *LNCS*, pages 1–30. Springer, 2011. doi:10.1007/978-3-642-35746-6_1.

16 Patrick Cousot. *Principles of Abstract Interpretation*. MIT Press, 2021. URL: https://mitpress.mit.edu/9780262044905/.

17 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL 1977*, pages 238–252. ACM, 1977. doi:10.1145/512950.512973.

18 Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of POPL 1979*, pages 269–282. ACM Press, 1979. doi:10.1145/567752.567778.

19 Patrick Cousot and Radhia Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6(1):69–95, 1999. doi:10.1023/A:1008649901864.

20 Patrick Cousot and Radhia Cousot. Temporal abstract interpretation. In *Proceedings of POPL 2000*, pages 12–25. ACM, 2000. doi:10.1145/325694.325699.

21 Patrick Cousot, Pierre Ganty, and Jean-François Raskin. Fixpoint-guided abstraction refinements. In *Proceedings of SAS 2007*, volume 4634 of *LNCS*, pages 333–348. Springer, 2007. doi:10.1007/978-3-540-74061-2_21.

22 Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming, Languages and Systems*, 19(2):253–291, 1997. doi:10.1145/244795.244800.

23 Dennis Dams and Orna Grumberg. Abstraction and abstraction refinement. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 385–419. Springer, 2018. doi:10.1007/978-3-319-10575-8_13.

24 Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order, Second Edition*. Cambridge University Press, 2002. doi:10.1017/CBO9780511809088.

25 Roberto Giacobazzi and Elisa Quintarelli. Incompleteness, counterexamples, and refinements in abstract model-checking. In *Proceedings of SAS 2001*, volume 2126 of *LNCS*, pages 356–373. Springer, 2001. doi:10.1007/3-540-47764-0_20.

26 Roberto Giacobazzi and Francesco Ranzato. Incompleteness of states w.r.t. traces in model checking. *Information and Computation*, 204(3):376–407, 2006. doi:10.1016/J.IC.2006.01.001.

27 Roberto Giacobazzi, Francesco Ranzato, and Francesca Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000. doi:10.1145/333979.333989.

28 Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Proceedings of CAV 1997*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997. doi:10.1007/3-540-63166-6_10.

29 Orna Grumberg, Martin Lange, Martin Leucker, and Sharon Shoham. When not losing is better than winning: Abstraction and refinement for the full mu-calculus. *Information and Computation*, 205(8):1130–1148, 2007. doi:10.1016/J.IC.2006.10.009.

30 Anubhav Gupta and Ofer Strichman. Abstraction refinement for bounded model checking. In *Proceedings of CAV 2005*, volume 3576 of *LNCS*, pages 112–124. Springer, 2005. doi:10.1007/11513988_11.

31 David Harel, Jerzy Tiuryn, and Dexter Kozen. *Dynamic Logic*. MIT Press, Cambridge, MA, USA, 2000. URL: https://mitpress.mit.edu/9780262527668/.

32 Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21:1–21:54, 2009. doi:10.1145/1592434.1592438.

33 Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko. Predicate abstraction for program verification. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 447–491. Springer, 2018. doi:10.1007/978-3-319-10575-8_15.

34 Dexter Kozen. Kleene algebra with tests. *ACM Transactions on Programming, Languages and Systems*, 19(3):427–443, 1997. doi:10.1145/256167.256195.

35 Kim Guldstrand Larsen and Bent Thomsen. A modal process logic. In *Proceedings of LICS 1988*, pages 203–210. IEEE Computer Society, 1988. doi:10.1109/LICS.1988.5119.

36 Hans Leiß. Towards Kleene algebra with recursion. In *Proceedings of CSL 1991*, volume 626 of *LNCS*, pages 242–256. Springer, 1991. doi:10.1007/BFB0023771.

37 Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995. doi:10.1007/BF01384313.

38 Damien Massé. Semantics for abstract interpretation-based static analyzes of temporal properties. In *Proceedings of SAS 2002*, volume 2477 of *LNCS*, pages 428–443. Springer, 2002. doi:10.1007/3-540-45789-5_30.

39 Antoine Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends in Programming Languages*, 4(3-4):120–372, 2017. doi:10.1561/2500000034.

40 Flemming Nielson and Hanne Riis Nielson. Model checking *is* static analysis of modal logic. In *Proceedings of FoSSaCS 2010*, volume 6014 of *LNCS*, pages 191–205. Springer, 2010. doi:10.1007/978-3-642-12032-9_14.

41 Flemming Nielson, Helmut Seidl, and Hanne Riis Nielson. A succinct solver for ALFP. *Nordic Journal of Computing*, 9(4):335–372, 2002.

42 Hanne Riis Nielson and Flemming Nielson. Flow logic: A multi-paradigmatic approach to static analysis. In *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday]*, volume 2566 of *LNCS*, pages 223–244. Springer, 2002. doi:10.1007/3-540-36377-7_11.

43 Peter W. O'Hearn. Incorrectness logic. *Proceedings of POPL 2020*, 4:10:1–10:32, 2020. doi:10.1145/3371078.

44 Francesco Ranzato. On the completeness of model checking. In *Proceedings of ESOP 2001*, volume 2028 of *LNCS*, pages 137–154. Springer, 2001. doi:10.1007/3-540-45309-1_10.

45 Francesco Ranzato and Francesco Tapparo. Making abstract model checking strongly preserving. In *Proceedings of SAS 2002*, volume 2477 of *LNCS*, pages 411–427. Springer, 2002. doi:10.1007/3-540-45789-5_29.

46 Francesco Ranzato and Francesco Tapparo. Generalized strong preservation by abstract interpretation. *Journal of Logic and Computation*, 17(1):157–197, 2007. doi:10.1093/LOGCOM/EXL035.

47 David Schmidt. Binary relations for abstraction and refinement. Technical report, Kansas State University, 2001.

48 David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proceedings of POPL 1998*, pages 38–48. ACM, 1998. doi:10.1145/268946.268950.

49 David A. Schmidt and Bernhard Steffen. Program analysis *as* model checking of abstract interpretations. In *Proceedings of SAS 1998*, volume 1503 of *LNCS*, pages 351–380. Springer, 1998. doi:10.1007/3-540-49727-7_22.

50 Sharon Shoham and Orna Grumberg. Monotonic abstraction-refinement for CTL. In *Proceedings of TACAS 2004*, volume 2988 of *LNCS*, pages 546–560. Springer, 2004. doi:10.1007/978-3-540-24730-2_40.

51 Bernhard Steffen. Data flow analysis as model checking. In *Proceedings of TACS 1991*, volume 526 of *LNCS*, pages 346–365. Springer, 1991. doi:10.1007/3-540-54415-1_54.

52 Bernhard Steffen. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming*, 21(2):115–139, 1993. doi:10.1016/0167-6423(93)90003-8.

53 Colin Stirling and David Walker. Local model checking in the modal mu-calculus. In *Proceedings of TAPSOFT 1989*, volume 351 of *LNCS*, pages 369–383. Springer, 1989. doi:10.1007/3-540-50939-9_144.

54 Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1/2):72–99, 1983. doi:10.1016/S0019-9958(83)80051-5.

55 Fuyuan Zhang, Flemming Nielson, and Hanne Riis Nielson. Model checking as static analysis: Revisited. In *Proceedings of IFM 2012*, volume 7321 of *LNCS*, pages 99–112. Springer, 2012. doi:10.1007/978-3-642-30729-4_8.