Equivalence-Based Abstraction Refinement for μHORS Model Checking

Xin Li and Naoki Kobayashi

The University of Tokyo, Tokyo, Japan
{li-xin,koba}@kb.is.s.u-tokyo.ac.jp

Abstract. Kobayashi and Igarashi proposed model checking of μHORS (recursively-typed higher-order recursion schemes), by which a wide range of programs such as object-oriented programs and multi-threaded programs can be precisely modeled and verified. In this work, we present a procedure for μHORS model checking that improves the procedure based on automata-based abstraction refinement proposed by Kobayashi and Li. The new procedure optimizes each step of the abstract-check-refine paradigm of the previous procedure. Specially, it combines the strengths of automata-based and type-based abstraction refinement as equivalence-based abstraction refinement. We have implemented the new procedure, and confirmed that it always outperformed the original automata-based procedure on runtime efficiency, and successfully verified all benchmarks which were previously impossible.

1 Introduction

The model checking of higher-order recursion schemes (HORS) can be considered as a generalization of finite-state and pushdown model checking, and has been recently applied to automated verification of higher-order programs [6, 10, 13]. A HORS [5, 12] is a simply-typed higher-order grammar that generates a possibly infinite tree called a value tree. The model checking problem for HORS is to check whether the value tree generated by HORS satisfies a given tree property. The problem is decidable [12] and a few efficient algorithms had been developed for it [14, 3], despite its extremely high worst-case complexity. Since HORS can be considered as a simply-typed higher-order functional program with recursion and tree constructors, the verification of functional programs (after predicate abstraction if necessary) can be naturally reduced to HORS model checking.

Although HORS can serve as a precise model for simply-typed higher-order functional programs, it could not be used for describing more expressive programs, such as functional programs with recursive types, object-oriented programs, and multi-threaded programs. To improve the expressiveness of HORS, Kobayashi and Igarashi introduced μHORS, an extension of HORS with recursive types, and studied a model checking problem for it [7]. Using μHORS, object-oriented programs and multi-threaded programs can be precisely modeled. They showed that μHORS model checking is undecidable, and developed a sound procedure for it based on the inference of recursive intersection types

that certify the safety of the grammar. The procedure is also relatively complete with respect to recursive intersection types: the grammar is eventually proved to be safe, if it is typable under some recursive intersection type system.

Kobayashi and Li later proposed an automata-based abstraction refinement procedure for μHORS model checking [8]. Following the *abstract-check-refine* paradigm, their procedure abstracts the configuration graph (that is the product reduction) of the grammar and the property automaton[6] as a finite graph, called *abstract configuration graph* (ACG), such that each term is abstracted as a state of a given finite tree automaton during the reduction, and the tree automaton is gradually refined for abstraction by counterexamples. Their procedure is sound and relatively complete with respect to a regular set of term trees: the grammar is eventually proved to be safe if there exists a regular set of term trees that is a safety inductive invariants for the grammar. Although their procedure is more efficient than the one proposed by Kobayashi and Igarashi, it is still not scalable enough as exhibited by their experiments.

To boost the scalability of μHORS model checking, we present a procedure that improves each step in the abstract-check-refine paradigm of the previous automata-based procedure as follows:

1) It combines the strengths of automata-based and type-based abstraction refinement as equivalence-based abstraction refinement: terms are identified to be equivalent during the reduction if and only if they are abstracted as the same state of the tree automaton and inhabit the same (non-recursive) intersection type; and such an equivalence relation is gradually refined by counterexamples.

2) The ACG construction is optimized so that the abstraction is more precise (i.e., the resulting ACG is in smaller size) than that of the original procedure, and therefore, our procedure is expected to scale better as confirmed by experiments.

3) An ACG is constructed by expanding different kinds of nodes in a specific order and by classifying the edges as two kinds of abstract and concrete reduction, respectively, so that the feasibility checking step (as to whether a counterexample is spurious or not) can be replaced by a simple and lightweight traversal of the error trace.

We have implemented the new procedure, and our empirical study showed that, it always outperformed the original procedure on runtime efficiency, and successfully verified all benchmarks from [8] which were previously impossible.

On the technical side, our new procedure also preserves the properties of the automata-based procedure, i.e., it is sound and relatively complete with respect to safety invariants in terms of a regular tree language. Note that, we are concerned with μHORS model checking in this work, but our techniques are applicable to the type-based abstraction refinement procedure for simply-typed HORS model checking [14].

The rest of the paper is organized as follows: Section 2 reviews μHORS model checking. Section 3 describes an improved procedure for μHORS model checking and its properties. Section 4 reports the implementation and experimental results. Section 5 discusses related work and Section 6 concludes the paper. The proofs of the theorems can be found in a full version of the paper [11].

2 Preliminaries

Let N_+ be the set of positive integers, and let E be a (finite) set. A tree D is a prefix-closed subset of N_+^* such that $\pi j \in D$ implies $\{\pi, \pi 1, \ldots, \pi(j-1)\} \subseteq D$, and an *$E$-labeled tree* is a map from a tree to E. We write $dom(f)$ for the domain of a map f. A *ranked alphabet Σ* is a map from a finite set of symbols to non-negative integers, such that $\Sigma(a)$ denotes the arity of each symbol $a \in dom(\Sigma)$. A *Σ-labeled ranked tree T* is a Σ-labeled tree satisfying that $T(\pi) = a$ implies $\{i \mid \pi i \in dom(T)\} = \{1, \ldots, \Sigma(a)\}$ for any $\pi \in dom(T)$. Here, an element $a \in dom(\Sigma)$ is used as a tree constructor.

The set of *recursive types*, ranged over by κ, is defined by:

$$\kappa \text{ (recursive types)} ::= \alpha \mid \kappa_1 \to \cdots \to \kappa_m \to \mathsf{o} \mid \mu\alpha.\kappa$$

where $m \geq 0$, α is a type variable, and $\mu\alpha.\kappa$ is an equi-recursive type with α bound by μ within the scope κ [2]. Here, $\to$ binds tighter than the binding operator μ. Intuitively, the type o represents (term) trees, and $\mu\alpha.\kappa$ represents a solution (that is a finite or infinite regular tree) to the type equation $\alpha = \kappa$, and therefore $\mu\alpha.\kappa = [\mu\alpha.\kappa/\alpha]\kappa$, e.g., $\mu\alpha.\alpha \to \mathsf{o}$ and $(\mu\alpha.\alpha \to \mathsf{o}) \to \mathsf{o}$ are identical. The type $\kappa_1 \to \cdots \to \kappa_m \to \mathsf{o}$ describes a function value that takes as arguments of type $\kappa_1, \ldots, \kappa_m$ and returns a tree. As usual, we call a type κ *closed* if all the type variables in κ are bound. We only consider closed types in the sequel.

Given a set of variables $\mathcal{V}$, a set of function symbols $\mathcal{F}$ that is disjoint with $\mathcal{V}$, and a ranked alphabet Σ, the set of *applicative terms* (or shortly terms), ranged over by t, is defined by: t (terms) $::= x \mid a \mid t_1 t_2$, where x ranges over $\mathcal{V} \cup \mathcal{F}$, and a ranges over $dom(\Sigma)$. A ground term is a term that contains no variables in $\mathcal{V}$.

A type environment $\mathcal{K}$ for recursive types is a map from $\mathcal{V} \cup \mathcal{F} \cup dom(\Sigma)$ to recursive types. The type judgement relation $\mathcal{K} \vdash t : \kappa$ is the least relation closed under the following rules:

$$\frac{}{\mathcal{K}, x : \kappa \vdash x : \kappa} \qquad \frac{}{\mathcal{K} \vdash a : \underbrace{\mathsf{o} \to \cdots \to \mathsf{o}}_{\Sigma(a)} \to \mathsf{o}} \qquad \frac{\mathcal{K} \vdash t_0 : \kappa_1 \to \kappa_2 \qquad \mathcal{K} \vdash t_1 : \kappa_1}{\mathcal{K} \vdash t_0 t_1 : \kappa_2}$$

Definition 1. *A μ**HORS** $\mathcal{G}$ is a tuple $(\mathcal{N}, \Sigma, \mathcal{R}, S)$, where*

– *$\mathcal{N}$ is a map from a set of non-terminal symbols to their recursive types.*

– *Σ is a ranked alphabet, and $dom(\Sigma)$ is called a set of terminal symbols.*

– *$\mathcal{R}$ is a set of rewriting rules in the form $F\,x_1 \cdots x_m \to t$ where F is a non-terminal symbol and $\mathcal{N}(F) = \kappa_1 \to \cdots \to \kappa_m \to \mathsf{o}$, and t is an applicative term such that $\mathcal{N}, x_1 : \kappa_1, \ldots, x_m : \kappa_m \vdash t : \mathsf{o}$. There exists exactly one rewriting rule for each non-terminal symbol F in $dom(\mathcal{N})$.*

– *$S \in dom(\mathcal{N})$ is called start symbol with $\mathcal{N}(S) = \mathsf{o}$.*

A μHORS [7] is a HORS [12] extended with recursive types, and can be considered a higher-order, call-by name, and recursively-typed functional program that generates a (possibly infinite) term tree.

Given a μHORS $\mathcal{G}$, the rewrite relation $\longrightarrow_{\mathcal{G}}$ on terms is defined by:

$$\frac{F\,\tilde{x} \to t \in \mathcal{R}}{F\,\tilde{s} \longrightarrow_{\mathcal{G}} [\tilde{s}/\tilde{x}]t} \qquad\qquad \frac{t_i \longrightarrow_{\mathcal{G}} t_i' \qquad i \in [1..\Sigma(a)]}{a\,t_1\,\cdots t_i\,\cdots\,t_{\Sigma(a)} \longrightarrow_{\mathcal{G}} a\,t_1\,\cdots t_i'\,\cdots\,t_{\Sigma(a)}}$$

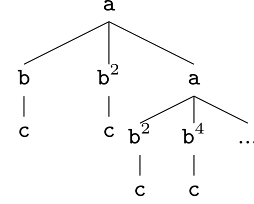
where $\tilde{x}$ and $\tilde{s}$ denote sequences of variables and terms, respectively.

Let $\bot \notin dom(\Sigma)$ be a fresh symbol. Let $\Sigma^{\bot}$ be the ranked alphabet that extends Σ with $\bot$ such that $\Sigma^{\bot}(\bot) = 0$. For a ground term t of type $\mathtt{o}$, we define the $\Sigma^{\bot}$-labeled ranked tree $t^{\bot}$ inductively as follows:

$$(a\,t_1\,\cdots\,t_{\Sigma(a)})^{\bot} = a\,(t_1{}^{\bot})\,\cdots\,(t_{\Sigma(a)}{}^{\bot}) \qquad (F\,\tilde{s}\,)^{\bot} = \bot$$

We define a partial order $\sqsubseteq$ on $\Sigma^{\bot}$-labeled ranked trees such that $C[\bot] \sqsubseteq C[t]$ for any tree t and tree context C. Let $\bigsqcup$ be the least upper-bound of trees with respect to $\sqsubseteq$. The *value tree* $\mathbf{Tree}(\mathcal{G})$ of $\mathcal{G}$ is a $\Sigma^{\bot}$-labeled tree $\bigsqcup\{t^{\bot} \mid S \longrightarrow_{\mathcal{G}}^{*} t\}$.

Example 1. Let $\mathcal{G}_1 = (\Sigma, \mathcal{N}, \mathcal{R}, S)$ where $\Sigma = \{\mathtt{a} \mapsto 3, \mathtt{b} \mapsto 1, \mathtt{c} \mapsto 0\}$,
$\mathcal{N} = \{S \mapsto \mathtt{o}, F \mapsto \mu\alpha.(\alpha \to (\mathtt{o} \to \mathtt{o}) \to (\mathtt{o} \to \mathtt{o}) \to \mathtt{o}), B \mapsto (\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}\}$,
$\mathcal{R} = \{S \to F\,F\,\mathtt{b}\,\mathtt{b}, B\,h\,x \to \mathtt{b}(h\,x)$
$\qquad F\,f\,k\,g \to \mathtt{a}\,(k\mathtt{c})\,(g(g\mathtt{c}))\,(f\,f\,(Bk)\,(Bg))\}$



S is reduced as follows:

$S \to F\,F\,\mathtt{b}\,\mathtt{b} \to \mathtt{a}\;(\mathtt{b}\,\mathtt{c})\,(\mathtt{b}^2\,\mathtt{c})\,(F\,F\,(B\,\mathtt{b})\,(B\,\mathtt{b})) \to \cdots$

$\mathbf{Tree}(\mathcal{G}_1)$ is shown to the right.

Definition 2. *A* ***trivial tree automaton*** *(TTA) $\mathcal{A}$ is a tuple (Σ, Q, δ, Q_0), where Σ is a ranked alphabet, Q is a set of states, $\delta \subseteq Q \times dom(\Sigma) \times Q^*$ satisfying that $m = \Sigma(a)$ if $(q, a, q_1 \cdots q_m) \in \delta$, and $Q_0 \subseteq Q$. Given a Σ-labeled ranked tree T. A* run tree *of $\mathcal{A}$ over T is a Q-labeled ranked tree R such that (i) $dom(R) = dom(T)$, (ii) $R(\epsilon) \in Q_0$, and (iii) $(R(\pi), T(\pi), R(\pi 1) \cdots R(\pi\Sigma(a)) \in \delta$ for any $\pi \in dom(R)$. $\mathcal{A}$* accepts *T if there is a run tree of $\mathcal{A}$ over T. We denote by $\mathcal{L}(\mathcal{A})$ the set of trees accepted by $\mathcal{A}$, and by $\mathcal{L}(\mathcal{A}, q)$ the set of trees accepted by the automaton $(\Sigma, Q, \delta, \{q\})$. $\mathcal{A}$ is top-down deterministic if (i) $|Q_0| = 1$ and (ii) $(q, a, q_1 \cdots q_m), (q, a, q_1' \cdots q_m') \in \delta$ implies $q_i = q_i'$ for each $i \in [1..m]$; and is moreover total if there exists $(q, a, q_1 \cdots q_{\Sigma(a)}) \in \delta$ for any $q \in Q$ and $a \in dom(\Sigma)$. We often write $\delta(q, a) = q_1 \cdots q_m$ for $(q, a, q_1 \cdots q_m) \in \delta$ when $\mathcal{A}$ is top-down deterministic. Dually, $\mathcal{A}$ is bottom-up deterministic if $(q, a, q_1 \cdots q_m), (q', a, q_1 \cdots q_m) \in \delta$ implies $q = q'$, and is total if there exists $(q, a, q_1 \cdots q_{\Sigma(a)}) \in \delta$ for any $q_1, \ldots, q_{\Sigma(a)} \in Q$ and $a \in dom(\Sigma)$.*

Trivial automata are originally considered by Aehlig [1] as non-deterministic Bϋchi tree automata where all the states are accepting. Note that, for finite trees, a topdown (resp bottom-up) deterministic TTA is just an ordinary topdown (resp bottom-up) deterministic finite tree automaton [4]. In this paper, we only consider topdown deterministic TTA.

We fix a μHORS $\mathcal{G} = (\mathcal{N}, \Sigma, \mathcal{R}, S)$ and a topdown deterministic TTA $\mathcal{A} = (\Sigma, Q, \delta, q_0)$ for the rest of paper. Let $\mathcal{A}^{\bot}$ denote the automaton $(\Sigma^{\bot}, Q, \delta \cup \{(q, \bot, \epsilon) \mid q \in Q\}, \delta, q_0)$. A *$\mu$HORS model checking problem* is to decide whether

$\mathbf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^{\perp})$. The μHORS model checking problem is undecidable [7], and we are concerned with sound and incomplete procedures for it.

Example 2. Let $\mathcal{A}_1$ be $(\Sigma, \{q_0, q_1, q_2, q_3\}, \delta, \{q_3\})$ where Σ is as given in Example 1, and δ is given as follows:

$$\{(q_3, \mathtt{a}, q_2 q_0 q_3), (q_0, \mathtt{b}, q_1), (q_1, \mathtt{b}, q_0), (q_2, \mathtt{b}, q_2), (q_2, \mathtt{c}, \epsilon), (q_0, \mathtt{c}, \epsilon), (q_3, \mathtt{c}, \epsilon)\}.$$

$\mathcal{A}_1$ accepts $\mathbf{Tree}(\mathcal{G}_1)$ in Example 1.

At the heart of practical procedures for higher-order model checking (e.g., [6, 8, 14]) is an algorithm for expanding a *configuration graph* of $\mathcal{G}$ and $\mathcal{A}$, starting with the root (S, q_0). A node in the graph is a pair (t, q) where t is a term and $q \in Q$ is a state of $\mathcal{A}$, and the edges obey the relation $\longrightarrow_{\mathcal{G},\mathcal{A}}$ defined by the following rules:

- $(F\, t_1 \cdots t_m, q) \longrightarrow_{\mathcal{G},\mathcal{A}} ([t_1/x_1, \ldots, t_m/x_m]s, q)$ if $F\, x_1 \cdots x_m \to s \in \mathcal{R}$.
- $(a\, t_1 \cdots t_m, q) \longrightarrow_{\mathcal{G},\mathcal{A}} (t_i, q_i)$ if $(q, a, q_1 \cdots q_m) \in \delta$ for $i \in [1..m]$.
- $(a\, t_1 \cdots t_m, q) \longrightarrow_{\mathcal{G},\mathcal{A}} \mathtt{fail}$ if $\delta(q, a)$ is undefined in $\mathcal{A}$.

Let $\longrightarrow^*_{\mathcal{G},\mathcal{A}}$ be the transitive and reflexive closure of $\longrightarrow_{\mathcal{G},\mathcal{A}}$.

Fact 3 $(S, q_0) \longrightarrow^*_{\mathcal{G},\mathcal{A}} \mathtt{fail}$ *if and only if* $\mathbf{Tree}(\mathcal{G}) \notin \mathcal{L}(\mathcal{A}^{\perp})$.

We consider the counterexample-guided abstraction refinement paradigm for model checking, and explore the following two finite means of guiding the abstraction refinement procedure: term automata and intersection types.

Term Automata [8]. A term automaton $\mathcal{B} = (\Sigma_{\mathcal{B}}, Q_{\mathcal{B}}, \delta_{\mathcal{B}}, q_{\mathcal{B},0})$ is a bottom-up deterministic and total finite tree automaton that accepts a regular set of *well-typed* ground term trees (with respect to the types of $\mathcal{G}$ on terminal and non-terminal symbols, and the type judgement relation $\mathcal{K} \vdash t : \kappa$ defined above).

We define an equivalence relation $\sim_{\mathcal{B}}$ on terms over $\Sigma_{\mathcal{B}}$ by, for any t and t', $t \sim_{\mathcal{B}} t'$ if and only if $\forall q \in Q_{\mathcal{B}}.\ t \in \mathcal{L}(\mathcal{B}, q) \Leftrightarrow t' \in \mathcal{L}(\mathcal{B}, q)$. That is, t and t' are equivalent if and only if they are accepted and rejected by the same states of $\mathcal{B}$.

Intersection Types. The higher-order model checking problem can be characterized as an intersection type inference problem [6, 9, 7]. Here, we limit our focus to non-recursive intersection types for rejection of the grammar by the complement of $\mathcal{A}$, and refer it shortly as intersection types or rejection types. The set of intersection types for $\mathcal{A}$ is given as follows:

$$\tau \text{ (strict types)} ::= q \mid \sigma \to \tau \qquad \sigma \text{ (intersection types)} ::= \bigwedge\{\tau_1, \ldots, \tau_k\}$$

where $q \in Q$ and $k \geq 0$. We write $\top$ for the empty intersection $\bigwedge \emptyset$. Note that, non-recursive intersection types are finitely many because the set of base types, i.e., the states of $\mathcal{A}$, is finite.

A type environment Γ is a set of type bindings in the form $h : \tau$, where $h \in \mathit{dom}(\mathcal{N}) \cup \mathit{dom}(\Sigma) \cup \mathcal{V}$. Note that, a type environment may have multiple type bindings for h. The type judgement $\Gamma \vdash_{\mathcal{A}} t : \tau$ is defined below following

[3]. Note that, since we are concerned with a top-down deterministic TTA as the property automaton, the rejection types used in our setting has a specific form.

$$\frac{}{\Gamma, x : \tau \vdash_{\mathcal{A}} x : \tau} \qquad \frac{\delta(q,a) = (q_1 \cdots q_m) \quad \forall i. i \in [1..m]}{\Gamma \vdash_{\mathcal{A}} a : \underbrace{\top \to \ldots \to \top}_{i-1} \to q_i \to \underbrace{\top \to \ldots \to \top}_{m-i} \to q}$$

$$\frac{\delta(q,a) \text{ is undefined}}{\Gamma \vdash_{\mathcal{A}} a : \underbrace{\top \to \ldots \to \top}_{\Sigma(a)} \to q} \qquad \frac{\Gamma \vdash_{\mathcal{A}} t_1 : \bigwedge\{\tau_1, \ldots, \tau_n\} \to \tau \quad \Gamma \vdash_{\mathcal{A}} t_2 : \tau_i \ (\forall i.\ i \in [1..n])}{\Gamma \vdash_{\mathcal{A}} t_1 t_2 : \tau}$$

For any term t, we define $\mathcal{T}_{[\Gamma]}(t) = \bigwedge\{\tau \mid \Gamma \vdash_{\mathcal{A}} t : \tau\}$.

We define an equivalence relation on terms by, for any terms t and t', $t \sim_\Gamma t'$ if and only if $\forall \tau. \Gamma \vdash_{\mathcal{A}} t : \tau \Leftrightarrow \Gamma \vdash_{\mathcal{A}} t' : \tau$. That is, t and t' are equivalent if they inhabit the same intersection types in Γ.

Example 3. Consider $\mathcal{A}_1$ in Example 2. We have $\mathcal{T}_{[\emptyset]}(a) = \bigwedge\{\tau_1, \tau_2, \tau_3, \tau_i \mid i \in \{0,1,2\}\}$. where $\tau_1 = q_1 \to \top \to \top \to q_3$, $\tau_2 = \top \to q_0 \to \top \to q_3$, $\tau_3 = \top \to \top \to q_3 \to q_3$, and $\tau_i = \top \to \top \to \top \to q_i$. $\mathcal{T}_{[\emptyset]}(b) = \bigwedge\{q_1 \to q_0, q_0 \to q_1, q_2 \to q_2, \top \to q_3\}$. $\mathcal{T}_{[\emptyset]}(c) = \bigwedge\{q_1\}$.

Definition 4. *An inductive invariant I for $\mathcal{G}$ is a set I of ground terms satisfying that, (i) $S \in I$; (ii) if $t \in I$ and $t \longrightarrow_{\mathcal{G}} t'$, then $t' \in I$. An inductive invariant I is regular if it is accepted by a finite tree automaton. A* ***safety invariant*** *for $\mathcal{G}$ (with respect to $\mathcal{A}$) is a regular inductive invariant I such that $t \in I$ implies $t^\perp \in \mathcal{L}(\mathcal{A}^\perp)$, i.e., I contains no invalid term trees [8].*

Fact 5 *If there exists a safety invariant for $\mathcal{G}$ wrt $\mathcal{A}$, then* $\mathbf{Tree}(\mathcal{G}) \in \mathcal{L}(\mathcal{A}^\perp)$.

A procedure for μHORS model checking is *sound* in the sense that, the grammar is safe if the procedure reports so, and *relatively complete* if the procedure eventually terminates and reports that the grammar is safe if there exists a safety invariant for $\mathcal{G}$ with respect to $\mathcal{A}$. The procedure may not terminate.

3 The Model Checking Procedure

We give an overview of the new procedure $MC_\sim(\mathcal{G}, \mathcal{A})$ in Figure 1 which depicts the high-level abstract-check-refine diagram explored in the procedure. The procedure takes as inputs a μHORS $\mathcal{G}$, a TTA $\mathcal{A}$, and an equivalence relation $\sim$ of a *finite* index on terms (i.e., $\sim$ induces a finite number of equivalence classes) which is used for directing the abstraction and refinement. Here, we combine the automata-based abstraction refinement [8] with the type-based approach [14], by taking $\sim = \sim_{\mathcal{B}} \cap \sim_\Gamma$.

Initially, $\sim_0 = \sim_{\mathcal{B}_0} \cap \sim_{\Gamma_0}$ provided with an initial term automata $\mathcal{B}_0$ [1] and $\Gamma_0 = \emptyset$. Starting with $\sim = \sim_0$, the procedure works as follows: The abstraction

[1] Note that, the choice of $\mathcal{B}_0$ would not affect relative completeness but practical efficiency of the procedure. An interested reader may wish to consult [8] for some approaches to constructing $\mathcal{B}_0$.

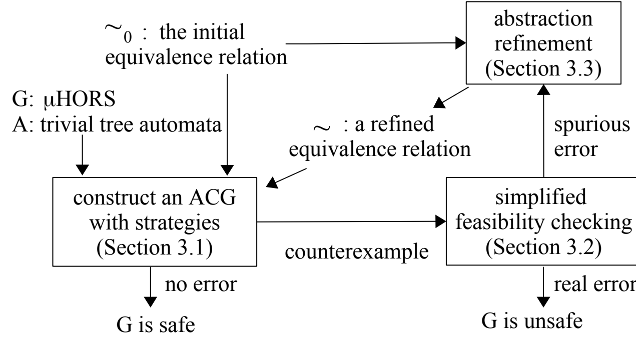


Fig. 1. Overview of the model checking procedure $MC_{\sim}(\mathcal{G}, \mathcal{A})$ for μHORS

step constructs a finite *abstract configuration graph* (ACG) as an abstraction of the configuration graph for $\mathcal{G}$ and $\mathcal{A}$, with *various strategies* (Section 3.1) and with the following twist that unifies [8] and [14]: any two nodes (t, q) and (t', q') are identified as equivalent and collapsed if and only if $t \sim t'$ and $q = q'$. Since $\sim$ has a finite index, there can be finitely many distinguished nodes in an ACG. If a closed ACG is constructed without containing any `fail` node, we conclude that the grammar is safe. Otherwise, a counterexample CE is raised during the ACG construction, and checked as to whether it is spurious or not (i.e., whether CE corresponds to a concrete reduction sequence that leads to `fail`). Thanks to the strategies applied to the ACG construction, this step is done by a simple and lightweight traversal of CE, called *simplified feasibility checking* (Section 3.2). If CE is a real error, we conclude that the grammar is unsafe. Otherwise, we refine the abstraction $\sim$ by refining $\sim_{\mathcal{B}}$ [8] and $\sim_{\Gamma}$ (Section 3.3), independently, so that the same CE would not occur in the future iterations. The loop is iterated until the grammar is proved or disproved. The procedure may not terminate since the model checking problem is undecidable in general.

3.1 Constructing Abstract Configuration Graph with Strategies

Overview of the Original ACG Construction. At the heart of model checking procedures in [8, 14] is an algorithm for constructing an ACG. Below we review the automata-based algorithm (i.e., take $\sim = \sim_{\mathcal{B}}$). Let $\mathcal{L}$ be a set of labels. A node in an ACG is either `fail` or a pair (t, q) of a state q in $\mathcal{A}$ and an *abstract applicative term* t given by: $t ::= a \mid F \mid x^{\ell} \mid t_1 t_2$, where $a \in dom(\Sigma)$, $F \in dom(\mathcal{N})$, and x^{ℓ} is an *abstract variable* annotated with a label $\ell \in \mathcal{L}$. Besides the graph, a map ρ is constructed from abstract variables to terms they are bound with. A map l_{α} from edges to reduction labels is also maintained.

Starting with the root (S, q_0), the algorithm *non-deterministically* and *fairly* takes a node N in the graph and expands it as follows (Here, by fairness, we mean any node to be expanded would be eventually chosen, so that if the grammar

is unsafe, an error trace would be eventually detected. It can be achieved for instance using an FIFO queue):

– Call $Expand_{\mathcal{N}}(N)$ for $N = (F\, s_1 \cdots s_m, q)$: given $F\, x_1 \cdots x_m \rightarrow t \in \mathcal{R}$, for each $i \in [1..m]$, an abstract variable $x_i^{\ell_i}$ is generated for representing the real argument s_i, where $\ell_i \in \mathcal{L}$ is fresh, and $\rho(x_i^{\ell_i}) = s_i$. A node $N' = ([x_1^{\ell_1}/x_1, \ldots, x_m^{\ell_m}/x_m]t, q)$ is generated and $l_\alpha(N, N') = F$.

– Call $Expand_{\Sigma}(N)$ for $N = (a\, s_1 \cdots s_m, q)$: if there exists $(q, a, q_1, \cdots, q_m)$ in δ, a node $N' = (s_i, q_i)$ is generated for each $i \in [1..m]$ and $l_\alpha(N, N') = (a, i)$; Otherwise, a `fail` node is generated.

– For each $x^{\ell'} \in dom(\rho)$, call $Expand_{\mathcal{V}}(N, (l, l'))$ for $N = (x^\ell\, s_1 \cdots s_m, q)$: a node $N' = (t\, s_1 \cdots s_m, q)$ is generated by replacing x^ℓ with the term $t = \rho(x^{\ell'})$ and $l_\alpha(N, N') = \varepsilon$. Besides, the edge (N, N') is labelled by (ℓ, ℓ').

For each abstract variable x^ℓ such that $\rho(x^\ell) = s$ for some s, we define a term $\rho^+(x^\ell) = [\rho^+(x_1^{\ell_1})/x_1^{\ell_1}, \ldots, \rho^+(x_n^{\ell_n})/x_n^{\ell_n}]s$, where $x_1^{\ell_1}, \ldots, x_n^{\ell_n}$ are variables occurring in s. Note that, a fresh label ℓ is always used in the construction. So the above equation cannot be circular and $\rho^+(x^\ell)$ is well defined. We extend the definition to any term t by $\rho^+(t) = [\rho^+(x_1^{\ell_1})/x_1^{\ell_1}, \ldots, \rho^+(x_n^{\ell_n})/x_n^{\ell_n}]t$, where $x_1^{\ell_1}, \ldots, x_n^{\ell_n}$ are variables in t.

Any two nodes $N = (C[x_1^{\ell_1}, \ldots, x_n^{\ell_n}], q)$ and $N' = (C[x_1^{\ell'_1}, \ldots, x_n^{\ell'_n}], q)$ are equivalent, denoted by $N \equiv N'$, if and only if $\rho^+(x_i^{\ell_i}) \sim \rho^+(x_i^{\ell'_i})$ for each $i \in [1..n]$. During the expansion, all $\equiv$-equivalent nodes are merged in the graph, by which an abstraction is applied to the reduction. The effect of the abstraction is reflected when expanding the variable-headed nodes. We call an ACG *closed* if no more nodes or edges can be added above. A closed ACG always exists and is finite, given that $\sim$ has a finite index.

Constructing an ACG with Strategies. Based on the original ACG construction above, Algorithm 1 constructs an ACG with various strategies for giving directions to the graph expansion:

1) A set $\mathcal{E}$ of *expandable label pairs* is constructed (line 35, 37), and it maintains the label pairs that can be used for expanding variable-headed nodes (line 12, 49), where $\mathcal{E}^\dagger$ denotes the disjoint union of $\mathcal{E}$ and $\{(\ell, \ell) \mid \ell \in \mathcal{L}\}$. Those labels pairs in $\mathcal{E}$ result from merging the node $N = (C[x_1^{\ell_1}, \ldots, x_n^{\ell_n}], q)$ with $N' = (C[x_1^{\ell'_1}, \ldots, x_n^{\ell'_n}], q)$ (line 34-35), defined by

$$EqLabels(N', N) = \{(\ell'_i, \ell_i) \mid \forall i \in [1..n].\ \ell_i \neq \ell'_i\}$$

When new expandable pairs are found, those related variable-headed nodes are expanded with more successors (line 10-15).

2) A specific order of expanding the graph is enforced using two worksets ws_0 and ws_1 for managing the nodes to be expanded. When taking a node from worksets (line 17-26), it always first takes a node from ws_0 if it is non-empty, and it takes a node from ws_1, otherwise. We classify edges $\longrightarrow_{\mathcal{C}}$ of the graph into two disjoint sets such that $\longrightarrow_{\mathcal{C}} = \longrightarrow_{\mathcal{C},1} \cup \longrightarrow_{\mathcal{C},0}$ (line 1-9).

Algorithm 1: Constructing an ACG with Strategies

```
1  proc Update(N, N', tag)
2  begin
3      if not tag then
4          →_{C,0} := →_{C,0} ∪ {(N, N')};
5          add N' to ws_0
6      else
7          →_{C,1} := →_{C,1} ∪ {(N, N')};
8          add N' to ws_1
9  end
10 proc NewExpand(E_new)
11 begin
12     foreach (ℓ, ℓ') ∈ E_new do
13         foreach N = (x^ℓ s̃, q) ∈ C do
14             N' := Expand_V(N, (l, l'));
15             Update(N, N', 1)
16 end
17 proc TakeNode(ws_0, ws_1)
18 begin
19     if ws_0 ≠ ∅ then
20         take N' from ws_0
21     else
22         if ws_1 ≠ ∅ then
23             take N' from ws_1
24         else raise an exception
25     return N'
26 end
27 →_{C,0} := ∅; →_{C,1} := ∅;
28 ws_0 := ∅; ws_1 := ∅; E := ∅;
29 add (S, q_0) to ws_0;
30 while ws_0 ≠ ∅ and ws_1 ≠ ∅ do
31     N := TakeNode(ws_0, ws_1);
32     InferType(C, N);
33     if N ≡ N' for some N' ≠ N ∈ C then
34         merge N with N';
35         E_new := EqLabels(N', N) \ E;
36         NewExpand(E_new);
37         E := E ∪ E_new;
38     else
39         if t = F s̃ then
40             N' := Expand_N(N);
41             Update(N, N', 0)
42         if t = a s̃ then
43             Succs := Expand_Σ(N);
44             foreach N' ∈ Succs do
45                 Update(N, N', 0);
46                 if N' = fail then
47                     return a counterexample
48         if t = x^ℓ s̃ then
49             foreach (ℓ, ℓ') ∈ E^† do
50                 N' := Expand_V(N, (l, l'));
51                 if ℓ = ℓ' then Update(N, N', 0)
52                 else Update(N, N', 1)
53 return the grammar is safe;
```

For nodes expanded from variable-headed nodes such that the head variable x^ℓ is replaced with the term $\rho(x^{\ell'})$ with $\ell \neq \ell'$, they are added to ws_1, and the resulting edges belong to $\longrightarrow_{\mathcal{C},1}$ (line 15, 52). For other nodes, they are added to ws_0 and the resulting edges belong to $\longrightarrow_{\mathcal{C},0}$ (line 41, 45, 51).

Enforcing expandable label pairs in $\mathcal{E}$ reduces redundant reduction sequences in an ACG that do not have any corresponding concrete reduction sequences. Thus, the abstraction becomes more precise. The advantage of 2) will be seen in Section 3.2 for simplifying the feasibility checking step.

Example 4. Recall $\mathcal{G}_1$ in Example 1 and $\mathcal{A}_1$ in Example 2. Assume $B\mathtt{b} \not\sim \mathtt{b}$, $BB\mathtt{b} \sim B\mathtt{b}$, and $B\mathtt{bc} \not\sim \mathtt{c}$ (e.g., $\sim = \sim_{\Gamma_0}$). Figure 2 shows a snapshot of part of an ACG for $\mathcal{G}_1$ and $\mathcal{A}_1$ without optimization, where for simplicity, we omit generating abstract variables for representing arguments of function calls to $B\,h\,x \rightarrow \mathtt{b}(h\,x)$ [2]. The binding relations are given as follows:

[2] It does not change the graph structure by doing so, because the arguments of B occurring in the reduction could never be merged according to the assumption on $\sim$

$$\rho(f^1) = F \qquad \rho(g^2) = \rho(k^3) = \mathtt{b} \qquad \rho(g^4) = B\,g^2$$
$$\rho(k^5) = B\,k^3 \qquad \rho(g^6) = B\,g^4 \qquad \rho(k^7) = B\,k^5$$

The node $(F f_1 (Bk^5)(Bg^4), q_3)$ has a child $(\mathtt{a}(k^7\mathtt{c})(g^6(g^6\mathtt{c}))(f^1 f^1(Bk^7)(Bg^6)), q_3)$, which is merged with the node $(\mathtt{a}(k^5\mathtt{c})(g^4(g^4\mathtt{c}))(f^1 f^1(Bk^5)(Bg^4)), q_3)$. If the graph is constructed by Algorithm 1, we have $\mathcal{E} = \{(5,7),(4,6)\}$ by merging the two nodes above, so that the entire subgraph circled by the dashed lines is not generated. Indeed, none of them has a corresponding concrete reduction.

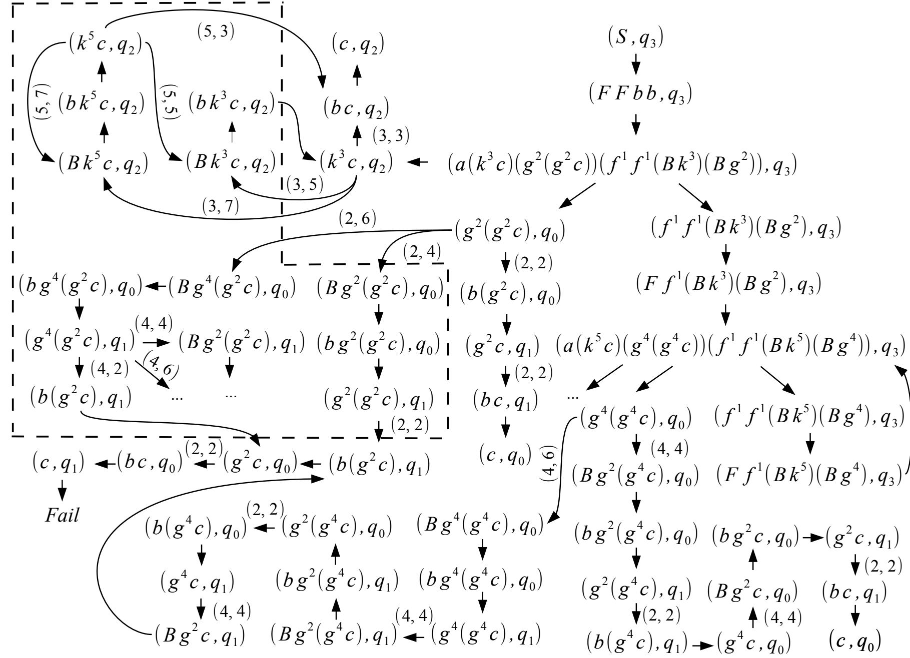


Fig. 2. A snapshot of an abstract configuration graph $\mathcal{C}$

3.2 Simplified Feasibility Checking

Given a counterexample CE, feasibility checking checks whether CE is spurious or not, i.e., whether there is a concrete reduction sequence that leads to `fail` by taking the same reduction labels along CE. When a cyclic CE is considered, it examines those (finite) abstract reduction sequences by unfolding CE up to a certain depth. Thanks to the order-guided ACG construction in Algorithm 1, we can conclude the following theorem, and the feasibility checking is replaced by a simple traversal of CE as to whether there exists an edge in $\longrightarrow_{\mathcal{C},1}$.

Theorem 1. *Let CE be the first counterexample raised by Algorithm 1. Then, (i) if there does not exist any edge on CE that belongs to $\longrightarrow_{\mathcal{C},1}$, then CE is a real error and the grammar is unsafe; and (ii) CE is spurious otherwise.*

The claim (ii) in Theorem 1 does not hold in general if CE is raised by a non-deterministic algorithm for constructing an ACG. The key is that, the usage of ws_0 and ws_1 in Algorithm 1 ensures that, for the edge $N \longrightarrow_{\mathcal{C},1} N'$ in CE nearest to `fail`, the complete subgraph rooted with N only having edges in $\longrightarrow_{\mathcal{C},0}$ has been constructed. Since it does not contain `fail`, CE must be spurious.

Remark 1. By the first counterexample above, we mean that it is firstly encountered during the model checking procedure, by taking the error trace that can be visited first obeying the expansion order than other options if any.

Example 5. Recall the ACG shown in Figure 2, excluding the part enclosed by the dashed line. There is a counterexample CE leading to `fail`. Since CE contains an edge $((g^4(g^4\mathtt{c}), q_0) \longrightarrow_{\mathcal{C},1} ((B\, g^4(g^4\mathtt{c}), q_0)$ labelled with $(4, 6)$, we know it is spurious by Theorem 1.

3.3 Abstraction Refinement of $\sim\ =\ \sim_{\mathcal{B}} \cap \sim_{\Gamma}$

Fig. 3 gives a sub-procedure $\mathit{InferType}(\mathcal{C}, N)$ that infers rejection types from a counterexample CE, when type-based abstraction refinement is combined with automata-based procedure as called at line 32 in Algorithm 1. The procedure takes as inputs an open ACG $\mathcal{C}$ and the current node $N \in \mathcal{C}$ to be expanded in the graph. Similar to the notion defined in [14], we say (t, q) is Γ-*rejected* if $\Gamma \vdash_{\mathcal{A}} t : q$. If N is Γ-rejected, the procedure takes a counterexample CE from $\mathcal{C}$ (line 2). If CE does not contain any edge in $\longrightarrow_{\mathcal{C},1}$, then it is a real error (line 3-4). Otherwise, a rejecting trail is taken from CE (line 6) from which a type environment Γ' is computed (line 7). By separately refining $\sim_{\mathcal{B}}$ as in [8] (line 8), $\sim$ is refined and another round of model checking is triggered (line 9).

```
1: if N is Γ-rejected then {
2:   take CE = e0...en from C
     that leads from (S,q0) to N;
3: if {e0,...,en} ⊆ ⟶C,0 then
4:   return the grammar is unsafe;
5: else {
6:   take a rejecting trail σ from CE;
7:   infer a type environment Γ' from σ;
8:   refine B to be B' by CE [8];
9:   call MC~(G,A) with ~ = ~B' ∩ ~Γ';
10: }}
```

Fig. 3. $\mathit{InferType}(\mathcal{C}, N)$: A sub-procedure for rejection type inference where $\mathcal{C}$ is an open ACG, and $N = (t, q)$ is a node in $\mathcal{C}$ to be expanded.

Our choice of rejecting trail is similar to the rejecting region defined in [14] (that is a subgraph of an ACG in which each node reaches to a Γ-rejected leaf), except that we are concerned with an open graph whereas a closed ACG is required by [14] for their abstraction and refinement.

Definition 6. *A trail is an alternating sequence of vertices and edges of a graph that starts and ends with vertices. Given CE* $= e_0 \dots e_n$ *where* $N_0 = (S, q_0)$, $N_{n+1} = N$ *and* $e_i = (N_i, N_{i+1}) \in \longrightarrow_{\mathcal{C}}$ *for each* $i \in [0..n]$*. A rejecting trail* $\sigma = N_k e_k N_{k+1} e_{k+1} \dots N_{n+1}$ $(k \in [0..n{+}1])$ *for CE is the longest trail, satisfying,*

(a) $\{e_k, \dots, e_n\} \cap \longrightarrow_{\mathcal{C},1} = \emptyset$*; and*

(b) For any $j \in [k..n]$, $(N_j, N') \notin \longrightarrow_{\mathcal{C},1}$ *for any* N' *if* $N_j = (x^{\ell}\, \tilde{s}, q)$*.*

The first condition says that σ only contains edges in $\longrightarrow_{\mathcal{C},0}$, and the second condition requires that, for any variable-headed node N_j in σ, N_j does not have any open successor to be reduced in the graph since $\mathcal{C}$ is an open graph. Note that, σ is unique for a given CE and could be just the Γ-rejected node N.

Given the rejecting trail $\sigma = N_k e_k N_{k+1} e_{k+1} \dots N_{n+1}$, rejection types are inferred from σ similar to [14, 9]. Starting with the Γ-rejected node N_n, types are extracted backwards along the trail as follows: for each node $N_i = (h s_1 \dots s_m, q)$ in σ where $i \in [0..n-k]$ and $h \in \Sigma \cup dom(\mathcal{N}) \cup \mathcal{V}$, if $h \in dom(\mathcal{N}) \cup \mathcal{V}$ (i.e., if h is headed by a non-terminal or a variable), we have

$$\Gamma^{(j)}(h) = \Gamma^{(j+1)}(h) \wedge \tau_j \text{ with } \tau_j = \bigwedge \mathcal{T}_{[\Gamma^{(j+1)}]}(s_1) \rightarrow \dots \rightarrow \bigwedge \mathcal{T}_{[\Gamma^{(j+1)}]}(s_m) \rightarrow q$$

and otherwise, $\Gamma^{(j)}(h) = \Gamma^{(j+1)}(h)$ when h is a terminal symbol, where $j = n - i$ and $\Gamma^{(n+1)} = \Gamma$. For any variable or non-terminal h that do not appear in head positions of nodes in σ, their types keep unchanged, i.e., $\Gamma^{(j)}(h) = \Gamma^{(j+1)}(h)$.

Theorem 2. *Given* Γ *is computed by the procedure in Figure 3. For any node* $N = (t, q)$ *in the ACG,* N *is* Γ*-rejected implies that* $(\rho^{+}(t), q) \longrightarrow^{*}_{\mathcal{G},\mathcal{A}} \texttt{fail}$.

By Theorem 2, we can safely raise a counterexample once a Γ-*rejected* node N is found, with no need for expanding it.

Example 6. Recall the previous error path CE in Figure 2. Let $\Gamma = \emptyset$. The rejecting node is $(\texttt{c}, q_1)$, and the rejecting trail σ is the sequence from the node $(B g^4 (g^4 \texttt{c}), q_0)$ to the node $(\texttt{c}, q_1)$. By type inference, we have Γ':

$\{g^2 : q_1 \rightarrow q_0, g^4 : q_1 \rightarrow q_1, B : (q_1 \rightarrow q_0) \rightarrow q_1 \rightarrow q_1, B : (q_1 \rightarrow q_1) \rightarrow q_1 \rightarrow q_0\}$

which ensures that $B\,\texttt{b} \not\sim_{\Gamma'} B\,B\,\texttt{b}$, so that the grammar can be proved safe in the next iteration of model checking.

3.4 Properties of the Procedure

Given a closed ACG $\mathcal{C}$ constructed by Algorithm 1, and let $\mathbb{C}$ be the configuration graph (CG) for $\mathcal{G}$ and $\mathcal{A}$. We show that there exists a weak simulation relation between $\mathbb{C}$ and $\mathcal{C}$.

Let $\mathcal{C} = (Node_{\mathcal{C}}, \longrightarrow_{\mathcal{C}})$, where $Node_{\mathcal{C}}$ is a finite set of nodes and $\longrightarrow_{\mathcal{C}} = \longrightarrow_{\mathcal{C},0} \cup \longrightarrow_{\mathcal{C},1}$ is a set of edges. Let $GNode_{\mathcal{C}} \subseteq Node_{\mathcal{C}}$ be the set of nodes where for any (t, q) in $GNode_{\mathcal{C}}$, t is headed by a terminal or a non-terminal symbol, and let $VNode_{\mathcal{C}} = Node_{\mathcal{C}} \setminus GNode_{\mathcal{C}}$ be the set of variable-headed nodes. Let $\rightarrow_{\tau} = \longrightarrow_{\mathcal{C}} \cap (VNode_{\mathcal{C}} \times Node_{\mathcal{C}})$, and the reflexive and transitive closure of $\rightarrow_{\tau}$ is denoted by

$\to_\tau^*$. Let $\to_\alpha = \longrightarrow_\mathcal{C} \cap (GNode_\mathcal{C} \times Node_\mathcal{C})$. Let $\Rightarrow_\mathcal{C} \subseteq \to_\alpha \to_\tau^* \cap (GNode_\mathcal{C} \times GNode_\mathcal{C})$. Recall that $l_\alpha(N, N') = \epsilon$ for any $(N, N') \in \to_\tau$. We extend l_α to $\Rightarrow_\mathcal{C}$ by, for any $(N, N') \in \Rightarrow_\mathcal{C}$ where $N \to_\alpha N'' \to_\tau^* N'$, $l_\alpha(N, N') = l_\alpha(N, N'')$.

Let $\mathbb{C} = (Node_\mathbb{C}, \to_\gamma)$ for $\mathcal{G}$ and $\mathcal{A}$, where $Node_\mathbb{C}$ is the (possibly infinite) set of nodes, and $\to_\gamma \subseteq Node_\mathbb{C} \times Node_\mathbb{C}$ is the set of edges, respectively. Let l_γ be a map from edges in $\mathbb{C}$ to reduction labels as usual. We also write $M \xrightarrow{a}_\gamma M'$ if $M \to_\gamma M'$ and $a = l_\gamma(M, M')$.

Definition 7. *For any abstract term t, we define $h(t)$ as the least set of ground terms satisfying: (a) $h(x^\ell) \supseteq h(x^{\ell'})$ if $(\ell, \ell') \in \mathcal{E}$; (b) $h(x^\ell) \supseteq h(\rho(x^\ell))$; (c) $h(t_1 t_2) \supseteq \{t_1' t_2' \mid t_1' \in h(t_1), t_2' \in h(t_2)\}$; and (d) $h(a) \supseteq \{a\}$ for any $a \in dom(\Sigma) \cup dom(\mathcal{N})$. A binary relation $\preceq \subseteq Node_\mathbb{C} \times GNode_\mathcal{C}$ is defined by, for any node $M = (s, q)$ in $Node_\mathbb{C}$ and any node $N = (t, q')$ in $GNode_\mathcal{C}$,*

$$M \preceq N \text{ if and only if } s \in h(t) \text{ and } q = q'.$$

Definition 8. *A relation $R \subseteq Node_\mathbb{C} \times Node_\mathcal{C}$ is a* weak simulation *if for every $(M, N) \in R$, (i) $M \preceq N$, and (ii) for any node M' and for any a such that $M \xrightarrow{a}_\gamma M'$, there exists a node N' such that $N \Rightarrow_\mathcal{C} N'$, $(M', N') \in R$, and $l_\alpha(N, N') = l_\gamma(M, M')$. Let M_0 and N_0 be the unique entry nodes of a CG and an ACG, respectively. We say that the ACG* weakly simulates *the CG if there exists a weak simulation R such that $(M_0, N_0) \in R$.*

Theorem 3 (soundness). *$\preceq$ is a weak simulation.*

It immediately follows Theorem 3 that, if a closed ACG does not contain any `fail` nodes, then the grammar is safe.

Theorem 4 (relative completeness). *$MC_\sim(\mathcal{G}, \mathcal{A})$ terminates and verifies that the grammar is safe, if there exists a safety invariant for $\mathcal{G}$ wrt $\mathcal{A}$.*

4 Experiments

We have implemented a prototype of the optimized procedure based on the model checker MuHorSar[8], and the tool is written in OCaml. We use Z3 4.3.3 (`http://z3.codeplex.com/`) as the backend constraint solver for automata-based abstraction refinement. We have evaluated the tools on examples from two categories of applications, including verification problems of FJ (Featherweight Java) programs and that of multi-threaded boolean programs with recursion. We are concerned with checking safety properties of the target programs. For multi-threaded programs, we studied properties of mutual exclusion (e.g., the Peterson's algorithm), deadlock-freedom (e.g., for various solutions to the dining philosopher problem), and checking of assertion violation (e.g., for simplified variants of Bluetooth drivers). Most of examples are taken from [7, 8] with a few examples newly-added as negative instances. Due to space, we only show those examples that couldn't be verified efficiently by the original procedure.

An interested reader may wish to consult [7, 8] for details of those examples and safety properties that have been checked against them. All experiments were conducted on a machine having a Mac OS X v.10.9.2, 1.7 GHz Intel Core i7 processor and 8GB RAM.

Table 1. Results for verifying FJ programs

scheme	$\#\mathcal{G}$	$\#\mathcal{A}$	R	MuHorSar		MuHorSar^{+}		MuHorSar$^{+}_{\sim}$	
L-filter	122	1	Y	1.391	(6)	0.867		0.919	
L-risers	122	1	Y	1.402	(6)	0.877		0.916	
stack-br	39	1	Y	0.309	(13)	0.071	(1)	0.060	(1)
		3		0.391	(13)	0.063	(1)	0.066	(1)
		5		0.408	(13)	0.065	(1)	0.063	(1)
queue-br	61	1	Y	0.253	(2)	0.194		0.203	
		3		0.251	(2)	0.213		0.206	
		5		0.261	(2)	0.196		0.200	
nat	35	1	Y	17.723	(147)	0.110		0.122	

Table 2. Results for verifying multi-threaded Boolean programs with recursion

scheme	$\#\mathcal{G}$	$\#\mathcal{A}$	R	MuHorSar		MuHorSar^{+}		MuHorSar$^{+}_{\sim}$	
locks-e	103	5	N	0.168	(1)	0.155		0.139	
dining-e	135	5	N	2.948	(28)	0.541		0.406	
dining-sp-e	193	5	N	11.685	(97)	0.884		0.833	
bluetooth	129	1	N	2.484	(26)	2.722	(14)	0.947	(5)
bluetooth-v1	158	1	N	—		68.819	(141)	3.658	(9)
bluetooth-v2	166	1	N	—		13.820	(54)	1.869	(9)
plotter-e	90	4	N	0.278	(3)	0.221		0.181	
dining-tan-e	303	5	N	—		5.923	(7)	5.824	(5)
peterson-e	74	2	N	0.589	(4)	0.257		0.270	
locks	95	5	Y	0.742		0.222		0.238	
plotter	88	4	Y	0.204		0.226		0.314	
peterson	74	2	Y	3.548	(2)	0.477		0.662	
peterson-d	80	9	Y	—		1.514		2.138	
dekker	94	2	Y	—		0.447		0.657	
pc-monitor	71	5	Y	0.331		0.222		0.354	
pc-sp	111	5	Y	2.238		0.219		0.370	
dining-tan	303	5	Y	—		18.229		23.007	

The preliminary experimental results for comparing the verification time taken by MuHorSar with and without optimizations are summarized in Table 1 for verifying FJ programs and in Table 2 for verifying multi-threaded programs, respectively. The column "scheme" shows the names of the examples. The columns "$\#\mathcal{G}$" and "$\#\mathcal{A}$" show the number of rules of the schemes and

the size of the property automaton for each example, respectively. The column "R" gives the answer whether the property is satisfied (Y) or violated (N). The column "MuHorSar" gives the runtime taken by the original procedure. The column "MuHorSar$^+$" gives the runtime with optimizing the abstraction refinement in MuHorSar, like enforcing $\mathcal{E}$, etc. The column "MuHorSar$^+_\sim$" shows the runtime by further combining the procedure with type-based abstraction refinement. The runtime is given in seconds, or "—" for timeout which is set to be 3 minutes. The number enclosed by parentheses shows the number of required abstraction refinement iterations, and we omit to show it in the table when it is zero, i.e., no abstraction refinement is needed.

As shown in both tables, the new procedure effectively improves the runtime of the original procedure. In particular, it successfully verified all of the benchmarks that were previously impossible. We found that enforcing expandable label pairs by $\mathcal{E}$ is very effective in scaling-up the model checking procedure, expected by reducing a large portion of redundant reduction sequences in an ACG.

5 Related Work

This work is an optimization and improvement of the automata-based procedure for μHORS model checking proposed by Kobayashi and Li [8]. Their abstraction-refinement approach explores a finite tree automaton for abstracting and identifying term trees (as states of the automaton) for constructing the abstract configuration graph, and often outperforms the first procedure for μHORS model checking proposed in [7]. Their idea is inspired by the type-directed abstraction refinement approach in [14], but is different in achieving (relative) completeness. In fact, the type-based approach applied to simply-typed HORS model checking in [14] would not ensure the same relative completeness as that is achieved by the automata-based procedure in [8], if applied to μHORS model checking.

This work makes an attempt to further improve the line of work. We combine automata-based and type-based abstraction refinement as an equivalence-based abstraction refinement, to take strengths of both approaches. We also proposed various optimizations to improve each step of the abstract-check-refine paradigm. Our improvements target on μHORS model checking but the ideas are applicable to improve the state-of-the-art model checker Preface for simply-typed HORS as well [14]. Preface is the model checker for HORS that first reported to scale to recursion schemes of several thousand rules. We expect that our approaches, such as enforcing expandable label pairs to reduce the size of an ACG, distinguishing abstract and concrete reductions in an ACG and working with an open configuration graph, etc., would be useful for further improving its scalability.

6 Conclusion

We have proposed systematic approaches to improve the runtime efficiency of the automata-based abstraction refinement procedure for μHORS model checking.

First, our approach combines the existing work on automata-based and type-based abstraction refinement techniques for higher-order model checking [8, 14, 7]. Next, we propose techniques for improving each step of the abstract-check-refine paradigm explored by the procedure. The new model checking procedure preserves the soundness and relative completeness properties of the original automata-based procedure [8]. We have implemented the new procedure, and confirmed by empirical study on examples of μHORS that, it always outperforms the original μHORS model checker MuHorSar, and successfully verified all benchmarks that were previously impossible. We are concerned with μHORS model checking but our approaches are applicable to the state-of-the-art model checker Preface for simply-typed HORS [14], and we expect our approaches would be useful for improving its scalability as well.

Acknowledgment

We would like to thank anonymous referees for useful comments. This work was supported by JSPS Kakenhi 15H05706.

References

1. Aehlig, K.: A finite semantics of simply-typed lambda terms for infinite runs of automata. Logical Methods in Computer Science 3(3) (2007)
2. Amadio, R.M., Cardelli, L.: Subtyping recursive types. ACM Transactions on Programming Languages and Systems 15(4), 575–631 (September 1993)
3. Broadbent, C.H., Kobayashi, N.: Saturation-based model checking of higher-order recursion schemes. In: Rocca, S.R.D. (ed.) CSL 2013. LIPIcs, vol. 23, pp. 129–148. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2013)
4. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata` (2007), release October, 12th 2007
5. Knapik, T., Niwinski, D., Urzyczyn, P.: Higher-order pushdown trees are easy. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. Lecture Notes in Computer Science, vol. 2303, pp. 205–222. Springer (2002)
6. Kobayashi, N.: Model checking higher-order programs. Journal of the ACM 60(3) (2013)
7. Kobayashi, N., Igarashi, A.: Model checking higher-order programs with recursive types. In: Felleisen, M., Gardner, P. (eds.) Proceedings of ESOP 2013. Lecture Notes in Computer Science, vol. 7792. Springer (2013)
8. Kobayashi, N., Li, X.: Automata-based abstraction refinement for μHORS model checking. In: Proceedings of LICS 2015. pp. 713–724. IEEE Computer Society (2015)
9. Kobayashi, N., Ong, C.H.L.: A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In: Proceedings of LICS 2009. pp. 179–188. IEEE Computer Society Press (2009)
10. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: Hall, M.W., Padua, D.A. (eds.) PLDI 2011. pp. 222–233. ACM (2011)

11. Li, X., Kobayashi, N.: Equivalence-based abstraction refinement for μHORS model checking. Full version, available from the first author's web page (2016)
12. Ong, C.H.L.: On model-checking trees generated by higher-order recursion schemes. In: Proceedings of LICS 2006. pp. 81–90. IEEE Computer Society Press (2006)
13. Ong, C.H.L., Ramsay, S.: Verifying higher-order programs with pattern-matching algebraic data types. In: Ball, T., Sagiv, M. (eds.) POPL 2011. pp. 587–598. ACM Press (2011)
14. Ramsay, S., Neatherway, R., Ong, C.H.L.: An abstraction refinement approach to higher-order model checking. In: Jagannathan, S., Sewell, P. (eds.) POPL 2014. ACM (2014)
15. Rehof, J., Urzyczyn, P.: Finite combinatory logic with intersection types. In: Proceedings of TLCA 2011. Lecture Notes in Computer Science, vol. 6690, pp. 169–183. Springer (2011)

Appendix

A Proof of Theorem 1

Lemma 1. *For any node $N = (t, q)$ in an ACG $\mathcal{C}$, it satisfies that, if $N \longrightarrow^*_{\mathcal{C},0}$ $\mathtt{fail}$, then $(\rho^+(t), q) \longrightarrow^*_{\mathcal{G},\mathcal{A}} \mathtt{fail}$.*

Proof. It is not hard to show that, $(t, q) \longrightarrow_{\mathcal{C},0} (t', q')$ implies $(\rho^+(t), q) \longrightarrow^*_{\mathcal{G},\mathcal{A}}$ $(\rho^+(t'), q')$. By induction on t,

- case $t = x^\ell s_1 \dots s_k$: we have $(t, q) \longrightarrow_{\mathcal{C},0} (t', q)$ where $t' = \rho(x^\ell)s_1 \dots s_k$. By definition, $\rho^+(t) = \rho^+(x^\ell)\rho^+(s_1)\dots\rho^+(s_m) = \rho^+(t')$, which implies $(\rho^+(t), q) \longrightarrow^*_{\mathcal{G},\mathcal{A}} (\rho^+(t'), q)$.
- case $t = as_1 \dots s_k$: consider some i, such that $(t, q) \longrightarrow_{\mathcal{C},0} (s_i, q_i)$. We know by definition that $\rho^+(t) = a\rho^+(s_1)\dots\rho^+(s_k)$, which implies $(\rho^+(t), q) \longrightarrow^*_{\mathcal{G},\mathcal{A}}$ $(\rho^+(s_i), q_i)$.
- case $t = Fs_1 \dots s_k$: given that $Fx_1 \dots x_k \to s \in \mathcal{R}$, we have that $(t, q) \longrightarrow_{\mathcal{C},0}$ (t', q') where $t' = [s_1/x_1, \dots, s_k/x_k]s$. We know by definition that $\rho^+(t) = F\rho^+(s_1)\dots\rho^+(s_k)$, which implies that
$(\rho^+(t), q) \longrightarrow^*_{\mathcal{G},\mathcal{A}} ([\rho^+(s_1)/x_1, \dots, \rho^+(s_k)/x_k]s, q) = (\rho^+(t'), q)$.

By induction on the depth of $\longrightarrow^*_{\mathcal{C},0}$, we further have that $(t, q) \longrightarrow^*_{\mathcal{C},0} (t', q')$ implies $(\rho^+(t), q) \longrightarrow^*_{\mathcal{G},\mathcal{A}} (\rho^+(t'), q')$, whereby the lemma is proved.

Lemma 2. *Consider Algorithm 1. If a node N (taken from the worksets at line 31) is to be expanded, such that $N' \longrightarrow_{\mathcal{C},1} N$, then the subgraph rooted with N' that contains only edges in $\longrightarrow_{\mathcal{C},0}$ has already been fully constructed in the graph.*

Proof. By Algorithm 1 and the assumption of Lemma 2, we know N' is a variable-headed node and it is expanded during line 48-52 in Algorithm 1. The sibling N'' of N such that $N' \longrightarrow_{\mathcal{C},0} N''$ was once added to ws_0 at the same iteration when N was added to ws_1, and it has been expanded in earlier time

(since, N is taken from $\mathtt{ws}_1$ which implies that $\mathtt{ws}_0$ is now empty). By a similar analysis (by induction on the depth of $N' \longrightarrow^*_{\mathcal{C},0} N''$), we know that the subgraph rooted with N' that contains only edges in $\longrightarrow_{\mathcal{C},0}$ has already been fully constructed in the graph.

Proof of Theorem 1

- The claim (i) follows immediately Lemma 1.
- To show claim (ii), let (N, N') be the edge on CE that is labeled with some (ℓ, ℓ') where $\ell \neq \ell'$ and be such an edge nearest to $\mathtt{fail}$. Let $N = (t, q)$ and $N' = (t', q)$. By assumption, we trivially have that $(t', q) \longrightarrow^*_{\mathcal{C},0} \mathtt{fail}$ which implies that $(\rho^+(t'), q) \longrightarrow^*_{\mathcal{G},\mathcal{A}} \mathtt{fail}$ by Lemma 1. We also have that $(t, q) \not\longrightarrow^*_{\mathcal{C},0} \mathtt{fail}$ which implies that $(\rho^+(t), q) \not\longrightarrow^*_{\mathcal{G},\mathcal{A}} \mathtt{fail}$.
  Otherwise, if $(t, q) \longrightarrow^*_{\mathcal{C},0} \mathtt{fail}$, then it violates the assumption that CE is the first raised counterexample. By Lemma 2, the subgraph that is rooted with N and consists in only edges in $\longrightarrow_{\mathcal{C},0}$ has been already expanded, before (N, N') is added to the graph. So another counterexample say CE' will be raised before CE. Note that, CE' is different from CE by assumption. So we have that $\rho^+(t) \not\longrightarrow^*_{\mathcal{G},\mathcal{A}} \mathtt{fail}$.
  It means that the term $\rho^+(t)$ is a valid term tree but $\rho^+(t')$ is an invalid term tree. Then CE must be spurious.

$\square$

B Proof of Theorem 2

Recall the following notion [9, 14]: a type environment Γ is *co-consistent* wrt $(\mathcal{G}, \mathcal{A})$ if, for each $F : \tau \in \Gamma$ such that $F \in dom(\mathcal{N})$ and $F\,\tilde{x} \to t \in \mathcal{R}$, we have $\Gamma \vdash_{\mathcal{A}} \lambda\tilde{x}.t : \tau$ and there is a finite witness (type derivation tree) for it.

Consider the type inference algorithm in Section 3.3. Our choice of a rejecting trail over an open configuration graph is a restricted and conservative case of a rejecting region in [14], and the way we collect rejection types is standard as [14, 6]. So the soundness property (as well as the progress property) of the type-directed abstraction refinement in [14] applies to our setting, as summarized below in Fact 9.

Fact 9 *The following invariant holds on the procedure in Figure 3: If Γ is co-consistent wrt $(\mathcal{G}, \mathcal{A})$, then the updated Γ' is co-consistent wrt $(\mathcal{G}, \mathcal{A})$. Since $\Gamma = \emptyset$, initially, and it is trivially co-consistent wrt $(\mathcal{G}, \mathcal{A})$, we can conclude that Γ is co-consistent throughout the model checking procedure.*

Proof of Theorem 2 We show that a node $N = (t, q)$ is Γ-rejected implies that $(t, q) \longrightarrow^*_{\mathcal{C},0} \mathtt{fail}$. The assumption (t, q) is Γ-rejected means that $\Gamma \vdash_{\mathcal{A}} t : q$. By induction on t.

- case $t = a s_1 \ldots s_k$: (i) if $\delta(q, a)$ is undefined in $\mathcal{A}$, then $(t, q) \longrightarrow_{\mathcal{C},0} \mathtt{fail}$ and the claim trivially holds; (ii) otherwise, there exists some i such that $\Gamma \vdash_{\mathcal{A}} s_i : q_i$ and $(t, q) \longrightarrow_{\mathcal{C},0} (s_i, q_i)$.

– case $t = Fs_1 \dots s_k$: given $Fx_1 \dots x_k \to s$, by Fact 9, we have that $\Gamma \vdash_{\mathcal{A}} [s_1/x_1, \dots, s_k/x_k]s : q$, and $(t, q) \longrightarrow_{\mathcal{C},0} ([s_1/x_1, \dots, s_k/x_k]s, q)$.

Since Γ is co-consistent wrt $(\mathcal{G}, \mathcal{A})$, any term t such that $\Gamma \vdash_{\mathcal{A}} t : q$ has a finite witness of proof. It is not hard to see that $(t, q) \longrightarrow^*_{\mathcal{C},0}$ `fail`, eventually. Then the claim immediately follows the above statement and Lemma 1. □

C Proof of Theorem 3

Lemma 3. *Given a ground term t and a variable x^ℓ such that $t \in h(x^\ell)$. Then there exists some l' such that $\rho(x^{\ell'})$ is defined, $t \in h(\rho(x^{\ell'}))$, and $(l, l') \in \mathcal{E}^*$.*

Proof. It immediately follows Definition 7 by induction on the transitive and reflexive closure of $\mathcal{E}$. □

Lemma 4. *Assume that a node (t_2, q) is merged with an existing node (t_1, q) during the construction of the ACG. Then $h(t_2) \subseteq h(t_1)$.*

Proof. Let $t_1 = C[x_1^{\ell_1}, \dots, x_k^{\ell_k}]$ and $t_2 = C[x_1^{\ell'_1}, \dots, x_k^{\ell'_k}]$. If (t_2, q) is merged with (t_1, q) in the graph, then according to the algorithm construction, $(\ell_i, \ell'_i) \in \mathcal{E}$ for each $i \in [1..k]$ where $\ell_i \neq \ell'_i$. By Definition 7 (a), it holds that $h(x_i^{\ell_i}) \supseteq h(x_i^{\ell'_i})$, which implies that $h(t_2) \subseteq h(t_1)$. □

Lemma 5. *Given a ground term s and a variable-headed node $(t, q) \in VNode_{\mathcal{C}}$ in an ACG such that $s \in h(t)$. Then there exists some t' such that $(t', q) \in GNode_{\mathcal{C}}$, $s \in h(t')$ and $(t, q) \to^*_\tau (t', q)$.*

Proof. Let $t = x^\ell t_1 \dots t_k$. By $s \in h(t)$, we have that $s = s_0 s_1 \dots s_t$ such that (i) $s_0 \in h(x^\ell)$ and (ii) $\forall i \in [1..k].\ s_i \in h(t_i)$. By (i) and Lemma 3, there exists some l' such that $\rho(x^{\ell'})$ is defined, $s_0 \in h(\rho(x^{\ell'}))$, and $(l, l') \in \mathcal{E}^*$. Let $t' = \rho(x^{\ell'}) t_1 \dots t_k$. Then $s \in h(t')$ and $(t, q) \to (t', q)$. If $(t', q) \in GNode_{\mathcal{C}}$, then the lemma is proved. Otherwise, we repeat the above analysis given the ground term s and a variable-headed node $N' = (t', q) \in VNode_{\mathcal{C}}$ where $s \in h(t')$. Since the domain of the binding function is finite, and for any variable headed node $N = (x^\ell t_1 \dots t_k, q)$ in the graph and any $(\ell, \ell') \in \mathcal{E}^\dagger$, the algorithm will expand it and generate an edge $N \to^*_\tau N'$ where $N' = (\rho(x^{\ell'}) t_1 \dots t_k, q)$ (by line 14 and 50 in Algorithm 1), we will finally find such l'' and variable y such that $\rho(y^{\ell''})$ is defined, $s_0 \in h(\rho(y^{\ell''}))$, $(t', q) \in GNode_{\mathcal{C}}$, $s \in h(t')$ and $(t, q) \to^*_\tau (t', q)$ where $t' = \rho(y^{\ell''}) t_1 \dots t_k$. □

Proof of Theorem 3 First, $\preceq$ is not empty since $M_0 \preceq N_0$ for $M_0 = (S, q_0)$ and $N_0 = (S, q_0)$. Consider any $M = (s, q) \in Node_{\mathbb{C}}$ and $N = (t, q) \in Node_{\mathcal{C}}$ such that $M \preceq N$. By definition, $s \in h(t)$. By induction on the structure of s:

– $s = as_1 \dots s_k$ where $a \in dom(\Sigma)$ and $k = dom(\Sigma)$: we have that $t = at_1 \dots t_k$ and $\forall i.s_i \in h(t_i)$. Consider any $M' = (s_i, q_i)$ for some $i \in [1..k]$ and $M \to_\gamma M'$. By the algorithm construction, there exists some node $N' = (t_i, q_i)$ such that $N \to_\alpha N'$, and $l_\gamma(M, M') = l_\alpha(N, N')) = (a, i)$.

If N' is not merged with any existing node, then it follows immediately that $s_i \in h(t_i)$. Otherwise, if $N \to_\alpha N''$ and $N'' = (t_i, q_i)$ is merged with the existing node $N' = (t', q_i)$, then by Lemma 4, $h(t_i) \subseteq h(t')$ and then we still have $s_i \in h(t')$.
If $N' \in \mathit{GNode}_\mathcal{C}$, then $M' \preceq N'$. Otherwise, t_i is variable-headed, and by Lemma 5, there exists some $N'' = (t', q_i) \in \mathit{GNode}_\mathcal{C}$ such that $N' \to_\tau^* N''$ and $s_i \in h(t')$, and then $M' \preceq N''$. Besides, $l_\alpha(N, N') = l_\alpha(N, N'')$.

- $s = Fs_1 \dots s_k$ where $F \in \mathit{dom}(\mathcal{N})$: we have that $t = Ft_1 \dots t_k$ and $\forall i.s_i \in h(t_i)$. Let $Fx_1 \dots x_k \to u \in \mathcal{R}$. By the algorithm construction, $M \to_\gamma M'$ where $M' = (s', q)$ and $s' = [s_1/x_1, \dots, s_k/x_k]u$, and $N \to_\alpha N'$ where $N' = (t', q)$ and $t' = [x_1^{\ell_1}/x_1, \dots, x_k^{\ell_k}/x_k]u$, and $l_\gamma(M, M') = l_\alpha(N, N') = F$.
If N' is not merged with any existing node, then $h(x_j^{\ell_j}) \supseteq h(t_j)$ since $\rho(x_j^{\ell_j}) = t_j$ for each j, by Definition 7. Then $s' \in h(t')$. Otherwise, if it is the case that $N \to_\alpha N''$ and $N'' = (t'', q)$ is merged with $N' = (t', q)$, then by Lemma 4, $h(t'') \subseteq h(t')$ and we still have $s' \in h(t'') \subseteq h(t')$.
If $N' \in \mathit{GNode}_\mathcal{C}$, then $M' \preceq N'$. Otherwise, N' is variable-headed, and by Lemma 5, there exists some $N'' = (t'', q) \in \mathit{GNode}_\mathcal{C}$ such that $N' \to_\tau^* N''$ and $s_i \in h(t'')$, and then $M' \preceq N''$. Besides, $l_\alpha(N, N') = l_\alpha(N, N'')$.

So in either case, for any M' and for any a, such that $M \xrightarrow{a}_\gamma M'$, then there exists N' such that $N \Rightarrow_\mathcal{C} N'$, $M' \preceq N'$, and $l_\alpha(N, N') = l_\gamma(M, M')$. □

D Proof of Theorem 4

We show the new procedure $MC_\sim(\mathcal{G}, \mathcal{A})$ is relatively complete, i.e., $MC_\sim(\mathcal{G}, \mathcal{A})$ terminates and verifies that the grammar is safe, if there exists a safety invariant I for $\mathcal{G}$ with respect to $\mathcal{A}$. Let $\mathcal{B}_*$ be a term automaton that accepts such a safety invariant I.

For the original automata-based abstraction refinement procedure, relative completeness is achieved by (gradually increasing k) cloning for k copies the states and transitions of the initial tree automaton $\mathcal{B}_0$, denoted by $\mathcal{B}_0^k$, from which a refined automaton $\mathcal{B} \subseteq \mathcal{B}_0^k$ is extracted. It is ensured that each refinement iteration takes a different $\mathcal{B}$ so that the all those counterexamples occurred in the previous iterations will no longer occur in the following iterations. Since the choice of $\mathcal{B}$ is finitely many for each k, we will finally find a $\mathcal{B}_*$ such that $\mathcal{L}(\mathcal{B}_*) = I$, by gradually increasing k until $k = |\mathcal{B}_*|$.

The new procedure consists in abstraction refinement of the term automaton $\mathcal{B}$ and (non-recursive) rejection types, as a refinement of an equivalence relation $\sim = \sim_\mathcal{B} \cap \sim_\Gamma$. According to [15, 3], given a type environment Γ of rejection types, one can construct a finite (alternating) tree automaton $\mathcal{B}_\Gamma$ that accepts $\{t \mid \Gamma \vdash_\mathcal{A} t : q_0\}$ (recall that q_0 is the initial state of $\mathcal{A}$). Therefore, for the new procedure combining the type-based abstraction refinement, the refinement of $\sim$ can be seen as performed on the product automaton $\mathcal{B}$ of $\mathcal{B}_0$ and $\mathcal{B}_\Gamma$. For the same reason above, we will finally find a refined automaton $\mathcal{B}_*$ accepting I.