SAINT-PETERSBURG UNIVERSITY

On the rights of the manuscript

Kostiukov Iurii Olegovich

Automatic Inference of Inductive Invariants of Programs With Algebraic Data Types

Scientific speciality
2.3.5. Mathematical and software support for computers, complexes and computer networks

Dissertation for the degree of
Candidate of Physico-Mathematical Sciences

Translation from Russian

Scientific supervisor:
Doctor of Science, Docent
Dmitry Koznov

Saint Petersburg — 2023

Contents

Page

Page

Introduction

Subject relevance. With software systems becoming increasingly ubiquitous and integrated into various aspects of human life, the issue of software reliability grows more critical. The problem of programs quality is traditionally handled by the field of formal methods. Starting from the 1990s, a new chapter began in this field with the emergence of binary decision diagrams and symbolic model checking based on efficient SAT solvers. This advancement enabled the verification of systems with up to 10^{120} possible program states [1]. The SAT revolution has led to a decline in the development of static analyzers from scratch. Instead, they are increasingly constructed atop a verification stack, consisting of SAT solvers for propositional logic, SMT solvers built upon them for first-order logic theories, and ultimately, Horn solvers for inductive invariant inference. New approaches to static analysis yield many benefits for the industry. For instance, in the development of Windows 7 in 2008, approximately one-third of all identified errors were discovered by SAGE [2], a tool that relies on symbolic execution and extensively uses an SMT solver to check the reachability of program execution branches.

Data types are of great importance in formal methods because suitable formalizations are required to take into account data types in program verification. However, most of the research is aimed at supporting "classical" data types, such as integers and arrays. Less researched are the emerging, becoming more popular data types, such as *algebraic data types (ADT)*[1]. They are constructed recursively, by union and Cartesian product of types. With ADTs one can build linked lists, binary trees, and other complex data structures. ADTs are actively employed in functional languages such as Haskell and OCaml, being an alternative structure to enumerations and unions from C and C++. Furthermore, ADTs are increasingly adapted in modern programming languages used in the industry, for example, in Rust and Scala, as well as in the languages of smart contracts, for example, in Solidity [3].

Thus, verifying the correctness of programs that use ADTs becomes an urgent task. This task can be formalized, and its solution partially automated within the framework of deductive verification based on Floyd-Hoare logic [4; 5] or refinement types [6], as, for example, in Flux system [7] for Rust and Leon system [8] for

[1]Depending on the context, they are also referred to as *abstract data types*, *inductive data types* and *recursive data types*.

Scala. However, such approaches require the user to provide *inductive invariants* to prove the correctness of the program, and formulating them in practice is an extremely laborious task. Verification systems based on independent programming languages and supporting ADT, such as, Dafny [9], Why3 [10], Viper [11], F* [12], face the same problem. It should also be noted that algebraic data types underlie numerous interactive theorem provers (ITPs), such as Coq [13], Idris [14], Agda [15], Lean [16]. Methods for automating induction in such systems are typically limited to syntactic enumeration, and therefore, during the proof process, the user is forced to carry out a laborious activity of formulating a sophisticated induction hypothesis, which is as hard as to infer an inductive invariant.

Thus, these problems are reduced to the problem of automatic inductive invariant inference of programs with algebraic data types. In general, the problem can be formulated using *constrained Horn clauses (CHCs)* — a special type of logical formulae that allows to simulate program's operation precisely [17].

Since the problem of automatic inductive invariant inference reduces to the problem of finding a model for a system of constrained Horn clauses, tools for automatic search of such models (so-called Horn solvers) can be applied in various contexts of program verification [18; 19]. For instance, RustHorn [20] utilizes Horn solvers for verifying Rust programs, while SolCMC [21] is employed for verifying Solidity smart contracts.

There are efficient Horn solvers with ADT support, such as Spacer [22] and its descendant Racer [23], as well as Eldarica [24], HoIce [25], RCHC [26], VeriCaT [27]. Annual international competitions CHC-COMP [28] are held among Horn solvers, where a separate section is devoted to solving Horn clause systems with algebraic data types.

The solution to a satisfiable system of Horn clauses is typically represented in the form of a *symbolic model* [19], which is a model expressed using first-order logic formulas in the constraint language of the Horn clause system. Therefore, the class of all inductive invariants definable in the constraint language I will call *(classical) symbolic invariants*. For example, all Horn solvers which participated in the CHC-COMP competition over the past two years build classical symbolic invariants.

The problem with symbolic invariants in the context of algebraic data types is that the constraint language of ADTs *does not allow expressing most of the inductive invariants required to verify practical programs*. And if a safe program does not have any inductive invariant that is definable in the constraint language, no algorithm for

inductive invariant inference in this language will be able to construct an inductive invariant for it. This leads to the fact that *Horn solvers that build classical symbolic invariants do not terminate on most systems with algebraic data types.*

Terms of algebraic data types have a *recursive structure*. For example, a binary tree is either a leaf or a vertex with two descendants, which are also binary trees. Hence, the primary reason why the ADT constraint language is unable to express inductive invariants for many programs is its limitation in expressing recursive relations over terms of algebraic types.

Development of the subject. The issue of inexpressiveness of the constraint language is well-known within the scientific community, and several attempts have been made to address this problem.

In 2018, P. Ruemmer (Sweden) proposed a method for inductive invariant inference in an extension of the constraint language with a size function that counts the number of constructors in a term. This extension was developed within the Horn solver ELDARICA [24]. However, the problem with this approach is that any extension of the constraint language requires a substantial reworking of the entire procedure for inductive invariant inference. In 2022, an extension of the constraint language with catamorphisms was proposed as part of the RACER Horn solver (H. Govind, A. Gurfinkel, USA) [23]. Catamorphisms are recursive functions of a simple form. However, their approach requires the user to specify the catamorphisms that will be used to build the inductive invariant *in advance*, so this approach is not completely automatic.

Since 2018, a separate line of research has been pursued by E. De Angelis, F. Fioravant, and A. Pettorossi in Italy [29–32]. Their line of work focuses on methods for eliminating algebraic types from Horn systems by reducing them to systems based on simpler theories, such as linear arithmetic. This approach is implemented in VERICAT [27]. A limitation of these methods is the inability to recover the original system's inductive invariant from the inductive invariant of a simpler system.

In 2020, it was suggested and implemented in the RCHC Horn solver (T. Haudebourg, France) [26] to express inductive invariants of programs over ADTs using *tree automata* [33]. However, tree automata do not allow representing *synchronous relations*, such as equality and disequality of terms, so the proposed approach is often inapplicable to the simplest programs, where inductive invariants can be easily found by traditional methods.

The goal of this thesis is to propose new classes of inductive invariants for programs with algebraic data types and to develop automatic inference methods for them. To achieve this goal, we have posed the following **tasks**.

1. Create new classes of inductive invariants of programs with algebraic data types that can express recursive and synchronous relations.
2. Create methods for automatic inductive invariant inference in new classes.
3. Develop a prototype software implementation of the proposed methods.
4. Conduct an experimental comparison of the implemented tool against existing alternatives using a representative benchmark.

Research methodology and methods. The research methodology involves designing classes of inductive invariants that are applicable in practice and developing corresponding algorithms while leveraging existing results in the field. In the study, first-order logic is utilized, along with fundamental concepts from automata theory and formal languages, including tree automata, synchronous automata, automaton languages, and the pumping lemma. The prototype implementation of the theoretical results was performed in the F# language, as well as partially in C++ within the code base of the Racer Horn solver (included in the Z3 SMT solver).

Main contributions to be defended.

1. We propose an effective method for automatic inductive invariant inference using tree automata, which can express recursive relations in a larger number of real programs. This method is based on the finite model finding.
2. We present a method for automatic inductive invariant inference through program transformation and finite model finding within a difficult for automatic invariant inference class. This class, based on synchronous tree automata, can express recursive and synchronous relations.
3. We propose a class of inductive invariants based on a Boolean combination of classical invariants and tree automata, which can express recursive relations in real programs and yet has an efficient inference procedure. Furthermore, we suggest an effective method for collaborative inductive invariant inference within this class by inferring invariants in subclasses.
4. A theoretical comparison is conducted between the existing and proposed classes of inductive invariants. This comparison includes formulating and proving pumping lemmas for both the constraint language and the constraint language extended by the term size function.

5. A pilot software implementation of the proposed methods in the F# language was conducted within the RInGen tool. This implementation was then compared to existing methods using the widely accepted "Tons of Inductive Problems" benchmark of functional program verification tasks. The best of the proposed methods was able to solve 3.74 times more tasks than the best performing implementation of the existing tools within the time limit.

The scientific novelty of the obtained results is as follows.

1. For the first time, a class of inductive invariants based on the Boolean combination of classes of classical invariants and invariants based on tree automata has been proposed.
2. For the first time, a finite model finding based algorithm for inductive invariant inference for programs with algebraic data types is proposed.
3. A new algorithm for collaborative inference of combined inductive invariants based on off-the-shelf methods for inferring invariants for separate classes has been proposed.
4. For the first time, pumping lemmas for first-order languages in the signature of the theory of algebraic data types have been introduced and proven.

Theoretical significance. The thesis offers new approaches for the inductive invariant inference. Since these approaches are orthogonal to the existing ones, they can be applied to programs over other theories, such as the theory of arrays, and can also strengthen already existing approaches for inductive invariant inference. Another significant theoretical contribution is the adaptation of pumping lemmas to first-order languages: these lemmas pave the way to a fundamental study of the undefinability of inductive invariants in first-order languages and the design of new classes of inductive invariants.

Practical significance. The proposed methods can be applied in the development of static analyzers for languages with algebraic data types. Since inductive invariants approximate loops and functions, they allow the analyzer to correctly "cut off" entire spaces of unreachable program states and avoid getting "stuck" in loops and recursion. For example, the proposed methods can be useful in the development of verifiers and test coverage generators for languages such as Rust, Scala, Solidity, Haskell and OCaml. Since the proposed methods were implemented in the pilot software, the resulting Horn solver can also be used as the "core" of a static analyzer, for example, for the Rust language using the RustHorn framework.

Reliability of the obtained results is ensured by computer experiments on publicly accepted benchmark and formal proofs. The results obtained in the thesis are consistent with the results of other authors in the field of inductive invariant inference.

Research validation. The main results of the work were reported at the following scientific conferences and seminars: HCVS 2021 International Workshop (March 28, 2021, Luxembourg), Huawei Workshop (November 18-19, 2021, St. Petersburg), JetBrains Research Annual Internal Workshop (December 18, 2021, St. Petersburg), PLDI 2021 conference (June 23-25, 2021, Canada), Internal seminar of the Vienna Technical University (June 3, 2022, Austria), LPAR 2023 conference (June 4-9, 2023, Colombia).

In 2021 and 2022, the developed tool took respectively 2nd and 1st place in the international CHC-COMP competition, in the ADT track.

Publications. The main results of the thesis are presented in 4 publications, 2 of which are published in journals recommended by the HAC, 2 are published in periodical scientific journals indexed by Web of Science and Scopus, one of which is published in the PLDI conference proceedings, which has an A* rank, and one is published in the LPAR conference proceedings, which has an A rank.

The author's **personal contribution** in joint publications is distributed as follows. In the article [34], the author implemented a reduction of inductive invariant inference of functions over complex data structures to solving systems of Horn clauses. Additionally, the author designed experiments with existing Horn solvers. The co-authors proposed the idea and developed its theoretical aspects. In the works [35], the author conducted a theoretical comparison of classes of inductive invariants, proposed and proved pumping lemmas for first-order languages over ADT, implemented the proposed approach, and conducted experiments. The co-authors participated in the discussion of the main ideas of the paper and performed a review of existing solutions. In the article [36], the author's contribution lies in proposing and formally justifying a collaborative approach to invariant inference, implementing and evaluating it. The co-authors participated in the discussion of the paper presentation and performed a review of existing solutions. In the article [37], the author's contribution lies in the formal description of the theory of computing preconditions for programs with complex data structures. The co-authors participated in the discussion of the main ideas and implemented the approach.

Volume and organization of the thesis. The thesis consists of an introduction, 6 chapters, and a conclusion.

The full volume of the thesis is 103 pages, including 5 code listings, 6 figures and 4 tables. The reference list contains 127 items.

Chapter 1. Background

This chapter presents the key concepts and theorems for this thesis, and outlines the state of the research field at the time of writing. Section 1.2 contains a brief history of the problem of expressivity of inductive invariants — the key problem for this thesis. Section 1.3 defines the constraint language, first-order logic, and algebraic data types — key objects for the verification methods proposed in the thesis. Section 1.4 presents constrained Horn clause systems and shows their connection with the program verification. Formal tree languages, used to represent sets of algebraic data types terms, are presented in Section 1.5. Finally, Section 1.6 presents the conclusions of the background.

1.1 Brief History of Software Verification

The history of verification is typically started with negative results: Turing's halting problem (1936) [38] and Rice's theorem (1953) [39]. These results state that there does not exist a verifier, which halts on all inputs and only gives correct results. The first constructive efforts towards automatic program verification were made by R. W. Floyd (1967) [4] and C. A. R. Hoare (1969) [5]. These researchers devised approaches that reduced program verification to checking satisfiability of logical formulas. The first practical approach to verification, known as *model checking*, emerged in 1981 within the context of concurrent program verification [40]. Its essential limitation was the so-called state explosion problem [41]: the state space grows *exponentially* as the state dimension increases.

To solve this problem, K. McMillan proposed *symbolic model checking* in 1987, which was implemented in the SMV tool later in 1993 [1].

Since 1996, a shift towards representing sets of program states by SAT (SATisfiability) formulas of propositional logic has been made [42]. This led to the verification of systems containing up to 10^{120} states [1]. It became possible thanks to a new generation of SAT solvers like CHAFF [43], based on the Conflict Driven Clause Learning (CDCL) algorithm for satisfiability checking [44]. Based on CDCL, the CDCL(T) algorithm for testing the satisfiability of first-order logic formulas in different theories (satisfiability modulo theories, SMT) was proposed in 2002 [45];

it was designed specifically for formal methods problems. In 2002, the first SMT solver CVC [46] was implemented on top of the CHAFF SAT solver.

The emergence of efficient SAT and SMT solvers led to separation of logical conditions checking and the global verification process. In 1999, bounded model checking (BMC) was proposed [47]. This method builds a logical formula from the unwinding of the transition relation of the program and passes it to an external solver. Then, in 1995−2000, thanks to R.P. Kurshan and E. Clarke, the counterexample-guided abstraction refinement (CEGAR) method appeared [48; 49]. This method allowed for the verification of programs by iteratively building inductive invariants as abstractions and refining them using counterexamples to the inductiveness of the candidate program invariants. In 2003−2005, K. Macmillan proposed to build abstractions using *interpolants* of unsatisfiable formulas extracted from a logical solver [50; 51]. Interpolants, in fact, are local partial proofs of the correctness of the program.

In 2012, it was proposed to add a so-called *Horn solver* to the "verifier, SMT solver, SAT solver" stack. Horn solver is responsible for automatic inference of inductive invariants and counterexamples [18]. Thus, the role of the verifier was reduced to the syntactic reduction of the program to a Horn clause system, and the Horn solver became the "core" of the verification process. For example, CEGAR approach is implemented in the Horn solver ELDARICA. In 2014 P. Garg proposed the ICE approach based on supervised learning [52]. ICE is implemented in the Horn solvers HOICE and RCHC.

In 2011, A. R. Bradley proposed an approach called IC3/PDR (property-directed reachability) [53] for SAT-based hardware verification. By 2014, the approach was generalized for SMT-based software verification [54; 55]. The IC3/PDR approach enhances CEGAR by creating abstractions through the construction of inductive strengthenings of the specification, evenly distributing resources between the search for an inductive invariant and a counterexample. IC3/PDR is implemented in the Horn solvers SPACER [22] and RACER [23].

Thanks to efficient algorithms, Horn solvers are more and more applied in verification of real programs, such as smart contracts.

1.2 History of the Inductive Invariant Expressivity Problem

Following the emergence of Floyd-Hoare logic in 1967−1969 [4; 5], the question of the sufficiency of the proposed calculus for proving the correctness of all possible programs became substantial. The correctness of the calculus was proven early on, but for many years, the problem of its completeness, i. e., whether the proposed calculus is sufficient to prove the safety of all safe programs, remained unresolved. Dealing with this problem in 1978 S. A. Cook proved [56] the *relative* completeness of Hoare logic. The relative completeness limitation in the theorem was that all possible weakest preconditions of the program must be expressible in the constraint language. Since that time, examples of simple programs whose invariants are inexpressible in the constraint language have been accumulated [57]. Therefore, in 1987 A. Blass and Yu. Gurevich proposed to abandon first-order logic in favor of *existential fixed-point logic* [58; 59]. This logic is significantly more expressive than first-order logic, so the classical completeness theorem without relativeness limitation was proved for it.

Note that negated existential fixed-point logic formulas correspond to constrained Horn clause systems [19]. The latter thus allows to express all possible inductive invariants of programs, but they are not an *effective* representation: the problem of checking the satisfiability of systems of Horn clauses is generally undecidable. Therefore, the problem of the invariant expressivity has not vanished, but it transformed instead into the main problem of this thesis: *how to express and efficiently build solutions of constrained Horn clause systems?*

At the moment, various approaches to solve this problem in practice are proposed: from the transformation of clause systems into systems in which the existence of an expressible invariant is more likely (see the works 2015−2022 E. De Angelis, A. Pettorossi [29–32; 60; 61]), syntactic synchronizations of clauses [61; 62], to the inference of relational invariants (invariants for several predicates) [63; 64].

In fact, research in the field of *completeness of abstract interpretation* is devoted to the solution of the same problem. Abstract interpretation is an approach [65] for building correct-by-construction static analyzers. Incompleteness in abstract interpretation arises from the approximation of undecidable properties in a decidable abstract domain, e. g., in some fragment of the first-order logic.

In 2000, it was shown that the abstract domain can be automatically refined by the analyzer [66]. However, as shown by R. Giacobazzi et al. [67] in 2015, this can lead to an overly precise abstract domain, causing the analyzer to diverge.

Therefore, the most important step in the abstract interpreter design is to come up with an abstract domain which will work well for a specific class of tasks [68]. Recent works in the field [69; 70] study the accuracy of the analysis and *local completeness*: completeness with respect to a given set of traces.

1.3 Constraint Language

For an arbitrary set X, define the following sets: $X^n \triangleq \{\langle x_1, \ldots, x_n\rangle \mid x_i \in X\}$ and $X^{\leqslant n} \triangleq \bigcup_{i=1}^{n} X^i$.

1.3.1 Syntax and Semantics of the Constraint Language

A multisort first-order signature with equality is a tuple $\Sigma = \langle \Sigma_S, \Sigma_F, \Sigma_P\rangle$, where Σ_S denotes the set of sorts, Σ_F represents the set of functional symbols, and Σ_P is the set of predicate symbols, which includes a distinguished equality symbol $=_\sigma$ for each sort σ. The equality sort index will be omitted in the following sections. Each functional symbol $f \in \Sigma_F$ has arity $\sigma_1 \times \cdots \times \sigma_n \to \sigma$, where $\sigma_1, \ldots, \sigma_n, \sigma \in \Sigma_S$, and each predicate symbol $p \in \Sigma_P$ has arity $\sigma_1 \times \cdots \times \sigma_n$. Terms, atoms, formulas, closed formulas, and first-order language (FOL) sentences are defined as usual. The first-order language defined over the signature Σ will be called the *constraint language*, and the formulas in it Σ-formulas.

A multi-sort structure (model) $\mathcal{M}$ for signature Σ consists of nonempty domains $|\mathcal{M}|_\sigma$ for each sort $\sigma \in \Sigma_S$. For each functional symbol f with arity $\sigma_1 \times \cdots \times \sigma_n \to \sigma$ we assign the interpretation $\mathcal{M}[\![f]\!] : |\mathcal{M}|_{\sigma_1} \times \cdots \times |\mathcal{M}|_{\sigma_n} \to |\mathcal{M}|_\sigma$, and to each predicate symbol p with arity $\sigma_1 \times \cdots \times \sigma_n$ we assign the interpretation $\mathcal{M}[\![p]\!] \subseteq |\mathcal{M}|_{\sigma_1} \times \cdots \times |\mathcal{M}|_{\sigma_n}$. For every closed term t with sort σ, the interpretation $\mathcal{M}[\![t]\!] \in |\mathcal{M}|_\sigma$ is defined recursively in a natural manner.

A structure is called finite if all domains of all its sorts are finite, otherwise it is called infinite.

The satisfiability of a clause φ in a model $\mathcal{M}$ is denoted by $\mathcal{M} \models \varphi$ and is defined as usual. By writing $\varphi(x_1, \ldots, x_n)$ instead of φ we will emphasize that all free variables in φ are among $\{x_1, \ldots, x_n\}$. Next, $\mathcal{M} \models \varphi(a_1, \ldots, a_n)$ denotes that $\mathcal{M}$ satisfies φ on an evaluation that maps free variables to elements of corresponding domains $a_1, \ldots, a_n$ (variables are also associated with sorts). The universal closure of the formula $\varphi(x_1, \ldots, x_n)$ is denoted by $\forall\varphi$ and is defined as $\forall x_1 \ldots \forall x_n.\varphi$. If φ

has free variables, then $\mathcal{M} \models \varphi$ means $\mathcal{M} \models \forall\varphi$. A formula is called *satisfiable in a free theory* iff it is satisfiable in some model of the same signature.

1.3.2 Algebraic Data Types

An algebraic data type (ADT) is a tuple $\langle C, \sigma \rangle$ where σ is the sort of this ADT, and C is a set of functional symbols of constructors. ADTs are also referred to as *abstract data types*, *inductive data types*, and *recursive data types*. With ADTs one can define data structures such as lists, binary trees, red-black trees, and others.

Let $\langle C_1, \sigma_1 \rangle, \ldots, \langle C_n, \sigma_n \rangle$ be an ADT set such that $\sigma_i \neq \sigma_j$ and $C_i \cap C_j = \varnothing$ for $i \neq j$. Due to the focus of this work, we will further consider only signatures of the theory of algebraic data types $\Sigma = \langle \Sigma_S, \Sigma_F, \Sigma_P \rangle$, where $\Sigma_S = \{\sigma_1, \ldots, \sigma_n\}$, $\Sigma_F = C_1 \cup \cdots \cup C_n$ and $\Sigma_P = \{=_{\sigma_1}, \ldots, =_{\sigma_n}\}$. Since Σ has no predicate symbols other than equality symbols (which have fixed interpretations within each structure), there is a single Herbrand model $\mathcal{H}$ for Σ. The domain of the Herbrand model $\mathcal{H}$ is a tuple $\left\langle |\mathcal{H}|_{\sigma_1}, \ldots, |\mathcal{H}|_{\sigma_n} \right\rangle$, where each set $|\mathcal{H}|_{\sigma_i}$ is a set of all closed terms of sort σ_i. The Herbrand model interprets all closed terms as themselves, and therefore serves as the standard model for the theory of algebraic data types. A formula φ will be called *satisfiable modulo the ADT theory* iff $\mathcal{H} \models \varphi$.

The satisfiability of formulas in free theory, as well as in ADT theory, can be checked automatically by the so-called *SMT solvers*, such as Z3 [71], CVC5 [72] and PRINCESS [73], and by automated theorem provers (ATPs) such as VAMPIRE [74]. These tools allow separating the task of building proofs of program safety from the task of verifying such proofs, automating the latter task.

1.4 Constrained Horn Clause Systems

By constrained Horn clause (CHC) systems, one can represent programs and their specifications by means of logic. The task of verifying programs in different (from functional to object-oriented) programming languages can be reduced to the problem of checking the satisfiability of constrained Horn clause systems [19]. That is why we formulate and examine the problem of inductive invariant inference for programs in terms of CHC systems, which makes CHC systems the central concept of this thesis.

1.4.1 Syntax

Let $\mathcal{R} = \{P_1, \ldots, P_n\}$ be a finite set of predicate symbols with sorts from signature Σ. Such symbols are called *uninterpreted*. A formula C over a signature $\Sigma \cup \mathcal{R}$ is called a *constrained Horn clause* (CHC) if it has the following form:

$$\varphi \wedge R_1(\bar{t}_1) \wedge \ldots \wedge R_m(\bar{t}_m) \to H.$$

Here, φ is a *constraint* (a constraint language formula without quantifiers), $R_i \in \mathcal{R}$, and $\bar{t}_i$ is a tuple of terms. H called the *head* of the clause is either false $\bot$ (in which case the clause is referred to as a *query*) or an atomic formula $R(\bar{t})$ (in which case the clause is called a *rule for* R). In this case, $R \in \mathcal{R}$ and $\bar{t}$ is a tuple of terms. The set of all rules for $R \in \mathcal{R}$ is denoted by $rules\,(R)$. The premise of the implication $\varphi \wedge R_1(\bar{t}_1) \wedge \ldots \wedge R_m(\bar{t}_m)$ is called the *body* of the formula C and is denoted as $body\,(C)$.

A (constrained) Horn clause (CHC) system $\mathcal{P}$ is a finite set of constrained Horn clauses.

1.4.2 Satisfiability and Safe Inductive Invariants

Let $\overline{X} = \langle X_1, \ldots, X_n \rangle$ be a tuple of relations such that if predicate P_i has sort $\sigma_1 \times \ldots \times \sigma_m$, then $X_i \subseteq |\mathcal{H}|_{\sigma_1} \times \ldots \times |\mathcal{H}|_{\sigma_m}$. To simplify notation, the model extension $\mathcal{H}\{P_1 \mapsto X_1, \ldots, P_n \mapsto X_n\}$ will be written as $\langle \mathcal{H}, X_1, \ldots, X_n \rangle$ or just $\langle \mathcal{H}, \overline{X} \rangle$.

A Horn clause system $\mathcal{P}$ is said to be *satisfiable modulo theory* ADT (or *safe*) if there exists a tuple of relations $\overline{X}$ such that $\langle \mathcal{H}, \overline{X} \rangle \models C$ for all $C formulas \in \mathcal{P}$. In such a case, the tuple $\overline{X}$ is referred to as a *(safe inductive) invariant* of the system $\mathcal{P}$. Thus, by definition, a Horn clause system is satisfiable if and only if it has a safe inductive invariant.

As the inductive invariant $\overline{X}$ is a tuple of sets which are infinite for most CHC systems, a class of inductive invariants is typically fixed in order to make automatic inductive invariant inference feasible. Such classes are design in such a way, so that their elements are finitely expressible. This thesis is focused on classes of inductive invariants with this property.

Note three important types of Horn clause systems: systems with no inductive invariants (unsatisfiable ones), systems with only one inductive invariant, and systems with multiple (even infinite) inductive invariants. It is also noteworthy that if a certain algorithm is designed to infer inductive invariants within some

fixed class, it may be the case that the system is satisfiable, yet none of its inductive invariants lies in that class. This typically leads to nontermination of the algorithm on such a system.

By notation $\mathcal{P} \in \mathcal{C}$, where $\mathcal{P}$ is the name of an example CHC system *with one uninterpreted symbol*, and $\mathcal{C}$ is an inductive invariants class, we mean that the system $\mathcal{P}$ is safe and *some* its safe inductive invariant (the relation interpreting the only predicate) belongs to the class $\mathcal{C}$.

Definition 1 (Elem). A relation $X \subseteq |\mathcal{M}|_{\sigma_1} \times \cdots \times |\mathcal{M}|_{\sigma_n}$ is called *expressible in first-order ADT language* (or *elementary*) if there exists a Σ-formula $\varphi(x_1, \ldots, x_n)$ such that $(a_1, \ldots, a_n) \in X$ if and only if $\mathcal{H} \models \varphi(a_1, \ldots, a_n)$. The class of all elementary relations will be denoted by Elem. The invariants in this class are called elementary, as well as *classical symbolic invariants*.

Elementary Invariants with Term Size Constraints

A tool Eldarica [24] infers invariants of Horn clause systems over ADT in extension of the constraint language by term size constraints. Let us define the class of invariants expressible by the formulas of this language.

Definition 2 (SizeElem). The SizeElem signature can be obtained from the Elem signature by adding the Int sort, operations from Presburger arithmetic, and functional symbols $size_\sigma$ with arity $\sigma \to Int$. For brevity, we will omit the σ sign in the $size$ symbols.

The satisfiability of formulas with term size constraints is checked in the structure $\mathcal{H}_{size}$, obtained by joining the standard model of Presburger arithmetic with the Herbrand model $\mathcal{H}$ and the following natural interpretation of the size function:

$$\mathcal{H}_{size}[\![size(f(t_1, \ldots, t_n)]\!] \triangleq 1 + \mathcal{H}_{size}[\![t_1]\!] + \ldots + \mathcal{H}_{size}[\![t_n]\!].$$

For example, the size of the term $t \equiv cons\big(Z, cons(S(Z), nil)\big)$ in the joint structure is evaluated as follows: $\mathcal{H}_{size}[\![size(t)]\!] = 6$.

1.4.3 Unsatisfiability and Resolution Refutations

It is well known that the unsatisfiability of a Horn clause system can be witnessed by a resolution refutation.

Definition 3. A *resolution refutation* (refutation tree) of a CHC system $\mathcal{P}$ is a finite tree with vertices $\langle C, \Phi \rangle$, where

(1) $C \in \mathcal{P}$ and Φ is a $\Sigma \cup \mathcal{R}$-formula;
(2) the root of the tree contains the query C and a satisfiable Σ-formula Φ;
(3) each leaf contains a pair $\langle C, \mathit{body}\,(C) \rangle$, where $\mathit{body}\,(C)$ is a Σ-formula;
(4) each tree node $\langle C, \Phi \rangle$ has children $\langle C_1, \Phi_1 \rangle, \ldots, \langle C_n, \Phi_n \rangle$ such that:
   - $\mathit{body}\,(C) \equiv \varphi \wedge P_1(\overline{x}_1) \wedge \ldots \wedge P_n(\overline{x}_n)$;
   - $C_i \in \mathit{rules}\,(P_i)$;
   - $\Phi \equiv \varphi \wedge \Phi_1(\overline{x}_1) \wedge \ldots \wedge \Phi_n(\overline{x}_n)$.

Theorem 1. A Horn clause system has a resolution refutation iff it is unsatisfiable.

1.4.4 From Verification to Solving Horn Clause Systems

Tools that automatically check the satisfiability of a Horn clause system are called *Horn solvers* (CHC solvers). Typically, a Horn solver either returns an inductive invariant or a resolution refutation, although it may also return "unknown" or diverge.

The problem of program verification can be reduced to the problem of checking the satisfiability of a Horn clause system [18; 19]. Among approaches providing such a reduction, the most significant are the Floyd-Hoare logic for imperative programs [4; 5], as well as dependent types [75] and refinement types [6] for functional programs. There are many tools within which this reduction can be implemented, for example, LiquidHaskell [76] for the Haskell language, RCaml [77] for OCaml, Flux [7] for Rust, Leon [8] and Stainless [78] for the Scala language. For example, tools like RustHorn [20], a verifier for the Rust language, and SolCMC [21], a smart contract verifier for the Solidity language, are based on the above approaches. These tools directly apply Horn solvers with ADT support, such as Spacer and Eldarica.

1.5 Tree Languages

Various types of sets of ADT terms, viewed as tree languages, are studied within the field of formal languages as generalizations of string languages. In particular, the generalization of (string) automata to tree automata and their extensions, which typically have the properties of decidability and closure of basic language

operations (for example, testing for emptiness of language intersections), are studied [79–84]. For this thesis, various classes of tree languages are of interest because they can serve as classes of safe inductive invariants for programs that use ADTs.

1.5.1 Properties and Operations

In order to design an efficient invariant inference algorithm, one typically needs to build the class of invariants with the following properties: closure under Boolean operations, decidability of the tuple membership problem in the invariant, and decidability of the invariant emptiness check problem.

Definition 4 (Boolean closure). Let an operation $\bowtie$ be either $\cap$ (set intersection), or $\cup$ (set union), or $\setminus$ (set subtraction). A class of sets is said to be closed under the binary operation $\bowtie$ if for each pair of sets X and Y from the given class the set $X \bowtie Y$ also lies in the class.

Definition 5 (Decidability of membership). The problem of determining whether a tuple of closed terms belongs to a particular set of terms is *decidable within a given class of term sets* iff the set of pairs of tuples of closed terms $\bar{t}$ and elements i of this class, such that i expresses some set I, and $\bar{t} \in I$ holds, is decidable.

Definition 6 (Decidability of emptiness). The problem of determining the emptiness of a set is *decidable in the class of term sets* iff the set of class elements expressing the empty set is decidable.

1.5.2 Tree Automata

Tree automata generalize classical string automata to tree languages (term languages), preserving the decidability and closure of basic operations. Classical results for tree automata and their extensions are presented in the book [33].

Definition 7. A *(finite) tree n-automaton* over (alphabet) Σ_F is a tuple $\langle S, \Sigma_F, S_F, \Delta \rangle$, where S is a (finite) set of states, $S_F \subseteq S^n$ is a set of final states, and Δ is a transition relation with the rules of the following form:

$$f(s_1, \ldots, s_m) \to s.$$

Here the following notations are used: functional symbols are denoted by $f \in \Sigma_F$, their arity is denoted by $ar(f) = m$, and states are denoted by $s, s_1, \ldots, s_m \in S$.

An automaton is called *deterministic* if there are no rules in Δ with the same left-hand side.

Definition 8. A tuple of closed terms $\langle t_1, \ldots, t_n \rangle$ is *accepted* by an n-automaton $A = \langle S, \Sigma_F, S_F, \Delta \rangle$, if $\langle A[t_1], \ldots, A[t_n] \rangle \in S_F$, where

$$A\big[f(t_1, \ldots, t_m)\big] \triangleq \begin{cases} s, & \text{if } \big(f(A[t_1], \ldots, A[t_m]) \to s\big) \in \Delta, \\ \text{not defined}, & \text{otherwise.} \end{cases}$$

Automaton language of A, denoted by $\mathcal{L}(A)$, is the set of all tuples of terms accepted by automaton A.

Example 1. Let $\Sigma = \langle Prop, \{(\_ \wedge \_), (\_ \to \_), \top, \bot\}, \varnothing \rangle$ be a propositional signature. Consider an automaton $A = \langle \{q_0, q_1\}, \Sigma_F, \{q_1\}, \Delta \rangle$ with a set of transition relations Δ presented below.

$$\begin{array}{lll} q_1 \wedge q_1 \mapsto q_1 & q_1 \to q_0 \mapsto q_0 & \\ q_1 \wedge q_0 \mapsto q_0 & q_1 \to q_1 \mapsto q_1 & \bot \mapsto q_0 \\ q_0 \wedge q_1 \mapsto q_0 & q_0 \to q_0 \mapsto q_1 & \top \mapsto q_1 \\ q_0 \wedge q_0 \mapsto q_0 & q_0 \to q_1 \mapsto q_1 & \end{array}$$

The automaton A accepts only true propositional formulas without variables.

1.5.3 Finite Models

There is a one-to-one correspondence between finite models of free theory formulas and tree automata [85]. This correspondence gives the following procedure for building tree automata from finite models. Using the finite model $\mathcal{M}$, for each predicate symbol $P \in \Sigma_P$ an automaton $A_P = \langle |\mathcal{M}|, \Sigma_F, \mathcal{M}(P), \Delta \rangle$ is built; for all automata a common transition relation Δ is defined — for each $f \in \Sigma_F$ with arity $\sigma_1 \times \ldots \times \sigma_n \mapsto \sigma$ and for each $x_i \in |\mathcal{M}|_{\sigma_i}$ we set $\Delta\big(f(x_1, \ldots, x_n)\big) = \mathcal{M}(f)(x_1, \ldots, x_n)$.

Theorem 2. For any automaton A_P, the following holds:

$$\mathcal{L}(A_P) = \{\langle t_1, \ldots, t_n \rangle \mid \langle \mathcal{M}[\![t_1]\!], \ldots, \mathcal{M}[\![t_n]\!] \rangle \in \mathcal{M}(P)\}).$$

The practical value of this result is that building a tree automaton for the formula is equivalent to finding a finite model for it. Therefore, a number of tools

such as MACE4 [86], KODKOD [87], PARADOX [88], as well as CVC5 [89] and VAMPIRE [90] can be used for finding finite models of free theory formulas and, as a result, to automatically build tree automata.

Most of these tools implement SAT encoding: the finite domain and functions are encoded into a bit representation via a propositional logic formula, which is then passed to a SAT solver. Finite-model finders are applied in verification [91], as well as in first-order infinite model building [92].

1.6 Conclusions

Automatic inductive invariant inference plays a key role in formal methods, particularly in static analysis. Despite the fact that there are a number of well-developed methods for inferring inductive invariants, and new papers on this topic appear each year at various A* computer science and programming language conferences (such as POPL, PLDI, CAV, etc.), as well as annual competitions between corresponding tools, the following problem still remains open: how to express the inductive invariants of programs. The challenge of designing the best representation of invariants lies in expressing the invariants of *real life programs* on the one hand, while having an *efficient invariant inference* procedure on the other hand. This problem is even more critical in the context of algebraic data types, for which the classical methods of representing invariants are extremely inefficient; and if the invariant is not representable, then the inference algorithm for this representation will not terminate. This makes the research conducted in this thesis in-demand and relevant.

Chapter 2. Regular Invariant Inference

The main contribution of this chapter is a new method of automatic inductive invariant inference for systems over ADT using automated theorem provers. In Section 2.1, the method is presented and its correctness is proven for simplified Horn clause systems without constraints, and in Section 2.2, the method is extended to arbitrary constraint Horn clause systems. Section 2.3 considers the class of regular invariants that can be inferred using the proposed method. Section 2.4 describes how the proposed method can be applied to automatically infer regular invariants using finite model finders. Unlike classical elementary invariants, regular invariants based on tree automata can express recursive relationships, and in particular, arbitrary deep properties of algebraic terms. As stated in Section 2.2, the proposed method can also be combined with general-purpose automated theorem provers. The chapter is based on [35].

2.1 Inference for Horn Clause Systems without Constraints

The core idea of the method is as follows. If a Horn clause system over ADTs without constraints has a model in the free theory, then it is also satisfiable in the ADT theory, and the model corresponds to some ADT inductive invariant.

Example 2. Consider the following Horn clause system over the algebraic data type of Peano numbers. The system encodes the parity predicate for Peano numbers $even$, and the property that "no two consecutive natural numbers can be even at the same time".

$$x = Z \rightarrow even(x) \tag{2.1}$$

$$x = S(S(y)) \wedge even(y) \rightarrow even(x) \tag{2.2}$$

$$even(x) \wedge even(y) \wedge y = S(x) \rightarrow \bot \tag{2.3}$$

Although this simple system is safe, it does not have a classical symbolic invariant, as will be shown in Chapter 5.

This system can be rewritten into the following equivalent Horn clause system without constraints.

$$\begin{aligned} \top &\to even(Z) \\ even(x) &\to even(S(S(x))) \\ even(x) \wedge even(S(x)) &\to \bot \end{aligned}$$

It corresponds to the following formula in the free theory.

$$\begin{aligned} &\forall x.(\top \to even(Z)) \wedge \\ &\forall x.(even(x) \to even(S(S(x)))) \wedge \\ &\forall x.(even(x) \wedge even(S(x)) \to \bot) \end{aligned}$$

This formula is satisfied by the following finite model $\mathcal{M}$.

$$\begin{aligned} |\mathcal{M}|_{Nat} &= \{0,1\} \\ \mathcal{M}(Z) &= 0 \\ \mathcal{M}(S)(x) &= 1 - x \\ \mathcal{M}(even) &= \{0\} \end{aligned}$$

Lemma 1 (Soundness). Assume that a Horn clause system without constraints $\mathcal{P}$ with uninterpreted predicates $\mathcal{R} = \{P_1, \ldots, P_k\}$ is satisfied in some model $\mathcal{M}$, i. e., $\mathcal{M} \models C$ for all $C \in \mathcal{P}$. Let the following be true:

$$X_i \triangleq \{\langle t_1, \ldots, t_n \rangle \mid \langle \mathcal{M}[\![t_1]\!], \ldots, \mathcal{M}[\![t_n]\!] \rangle \in \mathcal{M}(P_i)\}.$$

Then $\langle \mathcal{H}, X_1, \ldots, X_k \rangle$ is an inductive invariant of $\mathcal{P}$.

Proof. All clauses have the form

$$\forall \overline{x}.C \equiv P_1(\overline{t}_1) \wedge \ldots \wedge P_m(\overline{t}_m) \to H.$$

Take some tuple of closed terms $\overline{x}$ with appropriate sorts. Then from $\mathcal{M} \models \forall C$, by the definition of X_i it follows that

$$\overline{t}_1 \in X_i \wedge \ldots \wedge \overline{t}_m \in X_m \to H',$$

where H' is the corresponding substitution for H. By the definition of the satisfiability of a Horn clause, it follows that

$$\langle \mathcal{H}, X_1, \ldots, X_k \rangle \models P_1(\overline{t}_1) \wedge \ldots \wedge P_m(\overline{t}_m) \to H.$$

□

Thus, from the finite model for the example above, we can build a set $X \triangleq \{t \mid \mathcal{M}[\![t]\!] = 0\} = \{S^{2n}(Z) \mid n \geqslant 0\}$, which is a safe inductive invariant of the original system.

2.2 Inference for Constrained Horn Clause Systems

Given a constrained clause system, an equisatisfiable clause system without constraints can be built as follows. Without loss of generality, we can assume that the constraint of each clause contains negations only over atoms. Term equality literals can be eliminated via unification [93], and each literal of the inequality of form $\neg(t =_\sigma u)$ is replaced by the atomic formula $diseq_\sigma(t, u)$. For each algebraic type (C, σ) we also introduce a new uninterpreted symbol $diseq_\sigma$ and add it to the set of relational symbols $\mathcal{R}' \triangleq \mathcal{R} \cup \{diseq_\sigma \mid \sigma \in \Sigma_S\}$.

Next, we build a system of clauses $\mathcal{P}'$ over $\mathcal{R}'$ from the system $\mathcal{P}$ as follows. For each algebraic type (C, σ) in $\mathcal{P}'$ we add the following clauses for $diseq_\sigma$:

$$\top \to diseq_\sigma(c(\overline{x}), c'(\overline{x}')) \text{ for all various constructors } c \text{ and } c' \in C \text{ of sort } \sigma$$

and

$$diseq_{\sigma'}(x, y) \to diseq_\sigma(c(\ldots, \underbrace{x}_{i\text{-th position}}, \ldots), c(\ldots, \underbrace{y}_{i\text{-th position}}, \ldots))$$

for all constructors c of sort σ, all i and all x, y of sort σ'.

For each sort $\sigma \in \Sigma_S$ we denote the diagonal set as $\mathcal{D}_\sigma \triangleq \{(x,y) \in |\mathcal{H}|^2_\sigma \mid x \neq y\}$.

It is well-known that universally quantified Horn clauses have the least model, which is the denotational semantics of the program modeled by the clause system [19]. The least model is the least fixed point of the program transition operator. From these facts, the following lemma is trivially implied.

Lemma 2. The least inductive invariant of the clauses for $diseq_\sigma$ is the tuple of relations $\mathcal{D}_\sigma$.

A simple consequence of the previous lemma is the following fact.

Lemma 3. For the Horn clause system $\mathcal{P}'$ obtained by the transformation described above, if $\langle \mathcal{H}, X_1, \ldots, X_k, Y_1, \ldots, Y_n \rangle \models \mathcal{P}'$ then $\langle \mathcal{H}, X_1, \ldots, X_k, \mathcal{D}_{\sigma_1}, \ldots, \mathcal{D}_{\sigma_n} \rangle \models \mathcal{P}'$ (relations Y_i and $\mathcal{D}_{\sigma_i}$ interpret predicate symbols $diseq_{\sigma_i}$).

Example 3. A CHC system $\mathcal{P} = \{Z \neq S(Z) \to \bot\}$ is transformed into the following system, $\mathcal{P}'$.

$$\begin{aligned} \top &\to diseq_{Nat}(Z, S(x)) \\ \top &\to diseq_{Nat}(S(x), Z) \\ diseq_{Nat}(x, y) &\to diseq_{Nat}(S(x), S(y)) \\ diseq_{Nat}(Z, S(Z)) &\to \bot \end{aligned}$$

The correctness of the transformation given in this section is proved in the following theorem.

Theorem 3 (Soundness). Let $\mathcal{P}$ be a Horn clause system, and $\mathcal{P}'$ be a clause system obtained by the described transformation. If $\mathcal{P}'$ is satisfiable in the free theory, then the original system $\mathcal{P}$ has an inductive invariant.

Proof. Without loss of generality, we can assume that each clause $C \in \mathcal{P}$ has the following form:

$$C \equiv u_1 \neq t_1 \wedge \ldots \wedge u_k \neq t_k \wedge R_1(\overline{u}_1) \wedge \ldots \wedge R_m(\overline{u}_m) \to H.$$

In $\mathcal{P}'$ this clause is transformed into the following clause:

$$C' \equiv diseq(u_1, t_1) \wedge \ldots \wedge diseq(u_k, t_k) \wedge R_1(\overline{u}_1) \wedge \ldots \wedge R_m(\overline{u}_m) \to H.$$

Thus, each sentence in $\mathcal{P}'$ does not contain constraints (because *diseq* rules also do not contain constraints), which means that by previous correctness Lemma 1 $\mathcal{P}'$ has some inductive invariant $\langle \mathcal{H}, X_1, \ldots, X_k, U_1, \ldots, U_n \rangle$. Then by Lemma 3 we have $\langle \mathcal{H}, X_1, \ldots, X_k, \mathcal{D}_{\sigma_1}, \ldots, \mathcal{D}_{\sigma_n} \rangle \models C'$ for each $C' \in \mathcal{P}'$. However, it is obvious that:

$$\langle \mathcal{H}, X_1, \ldots, X_k, \mathcal{D}_{\sigma_1}, \ldots, \mathcal{D}_{\sigma_n} \rangle \, [\![C']\!] = \langle \mathcal{H}, X_1, \ldots, X_k \rangle \, [\![C]\!].$$

This means that $\langle \mathcal{H}, X_1, \ldots, X_k \rangle \models C$ for each $C \in \mathcal{P}$, so $\langle X_1, \ldots, X_k \rangle$ is the desired inductive invariant of the original system. □

Using the method for invariant inference. Arbitrary automated theorem provers, e. g., saturation-based, such as VAMPIRE [94], E [95] and ZIPPERPOSITION [96], can be used as a backend to check the satisfiability of first-order formulas.

However, saturations do not provide an effective class of invariants, since even checking whether a tuple of closed terms belongs to the set expressed by a saturation is undecidable [97]. For this reason, possible saturation-based inductive invariant classes are not considered in this thesis. However, the study of their subclasses and automatic invariant inference procedures for them are promising.

The following sections discuss the specialization of the proposed method for inference of more specific regular invariants.

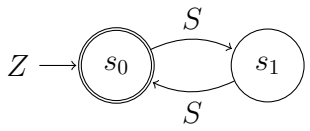
2.3 Regular Invariants

Definition 9 (Reg). We will say that an n-automaton A over Σ_F *expresses* the relation $X \subseteq |\mathcal{H}|_{\sigma_1} \times \ldots \times |\mathcal{H}|_{\sigma_n}$ if: $X = \mathcal{L}(A)$.

If for a relation X there exists a tree automaton expressing X, then the relation is called *regular*. The class of regular relations will be denoted as Reg.

Let $\mathcal{P}$ be a constrained Horn clause system. If $\overline{X} = \langle X_1, \ldots, X_n \rangle$ where every X_i is regular and $\langle \mathcal{H}, \overline{X} \rangle \models C$ for all $C \in \mathcal{P}$, then an inductive invariant $\langle \mathcal{H}, \overline{X} \rangle$ is called a *regular invariant* of $\mathcal{P}$.

Example 4. The Horn clause system from Example 2 has a regular invariant $\langle \mathcal{H}, \mathcal{L}(A) \rangle$, where A is a 1-tree automaton $\langle \{s_0, s_1, s_2\}, \Sigma_F, \{s_0\}, \Delta \rangle$, with the following transition relation Δ:



A set $\mathcal{L}(A) = \{Z, S(S(Z)), S(S(S(S(Z)))), \ldots\} = \{S^{2n}(Z) \mid n \geqslant 0\}$ trivially satisfies all clauses of the system.

Example 5. Consider the following clause system with a number of different invariants.

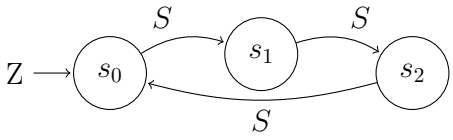
$$
\begin{aligned}
x = Z \wedge y = S(Z) &\rightarrow inc(x, y) \\
x = S(x') \wedge y = S(y') \wedge inc(x', y') &\rightarrow inc(x, y) \\
x = S(Z) \wedge y = Z &\rightarrow dec(x, y) \\
x = S(x') \wedge y = S(y') \wedge dec(x', y') &\rightarrow dec(x, y) \\
inc(x, y) \wedge dec(x, y) &\rightarrow \bot
\end{aligned}
$$

This system has an obvious elementary invariant

$$inc(x, y) \equiv (y = S(x)), dec(x, y) \equiv (x = S(y)).$$

This invariant is the strongest possible, since it expresses the denotational semantics of *inc* and *dec*. Yet these relations are not regular, i. e., there are no tree automata representing these relations [33].

However, this CHC system has a less obvious regular invariant based on two 2-tree automata $\langle \{s_0, s_1, s_2, s_3\}, \Sigma_F, S_*, \Delta \rangle$ with two sets of finite states, respectively, $S_{inc} = \{\langle s_0, s_1 \rangle, \langle s_1, s_2 \rangle, \langle s_2, s_0 \rangle\}$, $S_{dec} = \{\langle s_1, s_0 \rangle, \langle s_2, s_1 \rangle, \langle s_0, s_2 \rangle\}$ and with transition rules of the following form:



The automaton for *inc* predicate checks that $(x \, mod \, 3, \, y \, mod \, 3) \in \{(0,1), (1, 2), (2,0)\}$, and the automaton for *dec* checks that $(x \, mod \, 3, \, y \, mod \, 3) \in \{(1,0),(2,1),(0,2)\}$. These relations overapproximate the denotational semantics of *inc* and *dec* and prove the unsatisfiability of the formula $inc(x, y) \wedge dec(x, y)$. Therefore, although many relations might be not regular, programs still may have non-obvious regular invariants.

The properties of regular invariants are considered in more detail in Chapter 5.

2.4 Specialization for Regular Invariant Inference

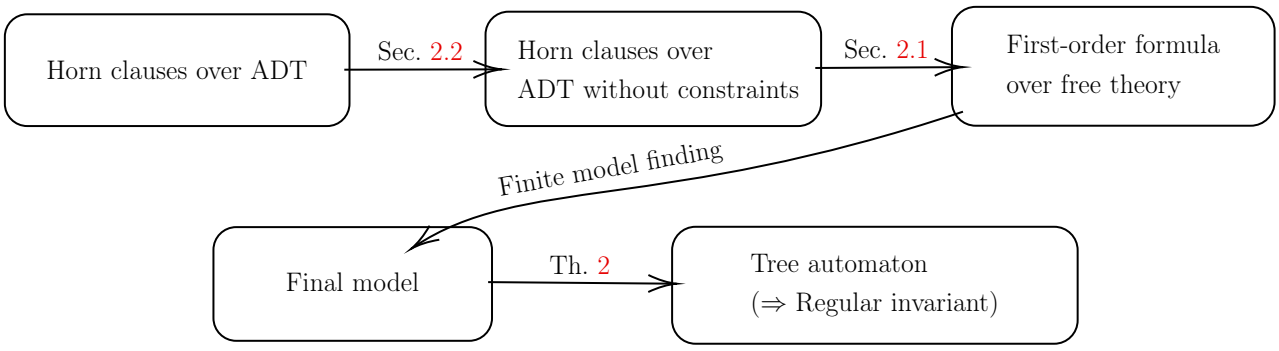
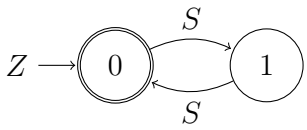


Figure 2.1 – Regular invariant inference method for a Horn clause system over ADT

The proposed method can be specialized to regular invariant inference, as shown in Figure 2.1. By employing the transformations from Sections 2.1 and 2.2,

it is possible to transform a constrained Horn clause system over ADTs into an equisatisfiable first-order formula over the free theory. If a finite model finder comes up with a finite model of this formula, then by application of a classical theorem 2 on isomorphism between finite models and tree automata it is possible to recover a tree automaton that expresses a regular invariant of the original Horn clause system. The correctness of the approach thus is ensured by Theorems 3 and 2.

For instance, from the finite model for the *Even* example from Section 2.1 we can obtain the following automaton A_{Even}, which is isomorphic to the one presented in Example 4.



In practice, this means that inductive invariants of constrained Horn clause systems over ADTs can be inferred automatically using *finite model finders*, such as Mace4 [86], Kodkod [87], Paradox [88], and general theorem provers, such as cvc5 [89] and Vampire [90], with appropriate options.

2.5 Conclusions

The proposed method reduces the problem of finding the inductive invariant of a constrained Horn clause system over ADTs to the problem of checking the satisfiability in a universal fragment of the first-order logic. Therefore, arbitrary automated theorem provers such as Vampire [94], E [95] and Zipperposition [96] can be used in combination with the proposed method for this task. These tools produce satisfiability proofs in the form of saturations, which can express a broad class of invariants. However, verifying whether a saturation represents an inductive invariant for a given CHC system is undecidable. Thus, using saturations to express inductive invariants is not feasible. Additionally, finite model finders can be utilized together with the proposed method. Examples of such tools include Mace4 [86], Kodkod [87], Paradox [88], and even cvc5 [89] and Vampire [90] in appropriate modes. The proposed method together with a finite model finder infers regular invariants based on tree automata that can express recursive relations and represent invariants for certain systems which do not have classical symbolic invariants. Moreover, checking that a given tree automaton expresses a regular invariant of a given system is decidable. A limitation of regular invariants is that they cannot represent

synchronous relations, such as increment of Peano integers or term equality. As a result, there are systems that have a classical symbolic invariant but lack regular ones. A richer *synchronous* regular invariant class that solves this problem, as well as a new method for inferring invariants for this class, are discussed in the next chapter.

Chapter 3. Synchronous Regular Invariant Inference

Synchronous tree automata are often used as an extension of tree automata capable of expressing synchronous relations. The expressive power of synchronous automata depends on the term convolution scheme on which this class is founded. Section 3.1 first discusses the class of synchronous regular invariants built upon synchronous automata with arbitrary convolution scheme. Then synchronous regular invariants based on full convolution, which can express a wide class of synchronous relations, are considered. Section 3.2 proposes a method for synchronous regular invariant inference, which is based on transforming a CHC system into a declarative description of a synchronous tree automaton that defines the invariant.

3.1 Synchronous Regular Invariants

Synchronous tree automata with standard [33] and full [26] convolutions are often viewed as a natural extension of classic tree automata for expressing synchronous relations, such as the equality and inequality of terms. In this section, we define tree automata with arbitrary convolution and prove their basic properties.

3.1.1 Synchronous Tree Automata

Definition 10. A term convolution is a computable bijective function from $\mathcal{T}(\Sigma_F)^{\leqslant k}$ to $\mathcal{T}(\Sigma_F^{\leqslant k})$ for some $k \geqslant 1$.

Definition 11 (cf. [26; 33]). The standard convolution of σ_{sc}-terms is defined as follows:

$$\sigma_{sc}\big(f_1(\overline{a}^1), \ldots, f_m(\overline{a}^m)\big) \triangleq \langle f_1, \ldots, f_m\rangle \left(\sigma_{sc}(\overline{a}_1^1, \ldots, \overline{a}_1^m), \sigma_{sc}(\overline{a}_2^1, \ldots, \overline{a}_2^m), \ldots\right).$$

Example 6. Consider the following application of the standard convolution to a tuple of terms:

$$\begin{aligned}\sigma_{sc}\big(n(p,q), S(Z), T(u,v)\big) &= \langle n, S, T\rangle \left(\sigma_{sc}(p, Z, u), \sigma_{sc}(q, v)\right)\\ &= \langle n, S, T\rangle \left(\langle p, Z, u\rangle, \langle q, v\rangle\right).\end{aligned}$$

Definition 12 (cf. [26]). The full convolution of σ_{fc}-terms is defined as follows:

$$\sigma_{fc}\big(f_1(\overline{a}^1), \ldots, f_m(\overline{a}^m)\big) \triangleq \langle f_1, \ldots, f_m\rangle \left(\sigma_{fc}(\overline{b}) \mid \overline{b} \in (\overline{a}^1 \times \ldots \times \overline{a}^m)\right).$$

Definition 13. A set of term tuples X is called a σ-*convolutional regular language* if there exists a tree automaton A such that $\mathcal{L}(A) = \{\sigma(\bar{t}) \mid \bar{t} \in X\} \triangleq \sigma(X)$.

The class of languages $\textsc{Reg}_\sigma$ is the set of all σ-convolutional regular languages. We denote by $\textsc{Reg}_+$ the class $\textsc{Reg}_{\sigma_{sc}}$ and by $\textsc{Reg}_\times$ the class $\textsc{Reg}_{\sigma_{fc}}$.

Lemma 4. Let L be a language of tuples of arity 1. Then it holds that $L \in \textsc{Reg}_\times \Leftrightarrow L \in \textsc{Reg}$.

Proof. By definition, we have $\sigma_{fc}\big(f(\bar{a})\big) \triangleq \langle f \rangle \left(\sigma_{fc}(\bar{b}) \mid b \in (\bar{a})\right)$. In other words, $\sigma_{fc}\big(f(a_1,\ldots,a_n)\big) \triangleq f\big(\sigma_{fc}(a_1),\ldots,\sigma_{fc}(a_n)\big)$. Therefore, $\sigma_{fc}(t) = t$ for all terms t, and hence $\sigma_{fc}(L) = L$ and $L \in \textsc{Reg}_\times$, $L = \sigma_{fc}(L) \in \textsc{Reg}$. □

Example 7. Consider the binary tree signature Σ_F with two constructors *Node* and *Leaf* (of arity 2 and 0, respectively), and the automaton $A = \left\langle \{\top,\bot\}, \Sigma_F^{\leqslant 2}, \{\bot\}, \Delta \right\rangle$ with the transition relation Δ:

$$\begin{aligned} \mathit{Leaf} &\to \bot & \langle \mathit{Node}, \mathit{Node}\rangle(\varphi,\psi) &\to \varphi \wedge \psi \\ \mathit{Node}(\varphi,\psi) &\to \bot & \langle \mathit{Node}, \mathit{Leaf}\rangle(\varphi,\psi) &\to \bot \\ \langle \mathit{Leaf}, \mathit{Leaf}\rangle &\to \top & \langle \mathit{Leaf}, \mathit{Node}\rangle(\varphi,\psi) &\to \bot, \end{aligned}$$

where φ and ψ range over all possible states. This automaton expresses the inequality relation using standard convolution. In other words, $\mathcal{L}(A) = \{\sigma_{sc}(x,y) \mid x,y \in \mathcal{T}(\Sigma_F), x \neq y\}$.

Example 8 (*lt*). Consider the signature Σ_F of Peano integers, which has two constructors Z and S (arity 0 and 1, respectively), and the following set, which defines an order on numbers:

$$\mathit{lt} \triangleq \Big\{ \big(S^n(Z), S^m(Z)\big) \mid n < m \Big\}.$$

Consider automaton $A = \left\langle \{\bot,\top\}, \Sigma_F^{\leqslant 2}, \{\top\}, \Delta \right\rangle$ with transition relation Δ:

$$\begin{aligned} \langle Z, Z\rangle &\to \bot & \langle Z, S\rangle(\varphi) &\to \top \\ Z &\to \bot & \langle S, Z\rangle(\varphi) &\to \bot \\ S(\varphi) &\to \bot & \langle S, S\rangle(\varphi) &\to \varphi, \end{aligned}$$

where $\varphi \in \{\top,\bot\}$ ranges over all possible states. This automaton expresses the order relation using standard convolution. In other words, $\mathcal{L}(A) = \{\sigma_{sc}(S^n(Z), S^m(Z)) \mid n < m\}$.

3.1.2 Closure Under Boolean Operations

Convolutional regular languages are closed under all Boolean operations regardless of the convolution. The proofs and corresponding constructions for classical tree automata essentially apply to convolutional regular languages. In this section, we will denote by k the tuple dimension of languages from REG_σ.

Theorem 4. The class of languages REG_σ with arbitrary convolution σ is closed under complement.

Proof. Let language $L \in \mathrm{REG}_\sigma$. Then without loss of generality we can say that there exists a deterministic automaton $A = \langle S, \Sigma_F^{\leqslant k}, S_F, \Delta \rangle$ such that $\mathcal{L}(A) = \sigma(L)$. Consider the automaton for the complement language $A^c = \langle S, \Sigma_F^{\leqslant k}, S \setminus S_F, \Delta \rangle$. It is true that $\mathcal{L}(A^c) = \overline{\mathcal{L}(A)} = \overline{\sigma(L)} = \sigma(\overline{L})$ (the latter follows from the fact that σ is a bijective function). Thus, we have $\overline{L} \in \mathrm{REG}_\sigma$. □

Theorem 5. The class of languages REG_σ with arbitrary convolution σ is closed under intersection.

Proof. Consider $L_1, L_2 \in \mathrm{REG}_\times$. Then we have deterministic automata $A = \langle S^A, \Sigma_F^{\leqslant k}, S_F^A, \Delta^A \rangle$ and $B = \langle S^B, \Sigma_F^{\leqslant k}, S_F^B, \Delta^B \rangle$ such that $\mathcal{L}(A) = L_1$ and $\mathcal{L}(A) = L_2$. Intersection of languages $L_1 \cap L_2$ is recognized by an automaton

$$C = \langle S^A \times S^B, \Sigma_F^{\leqslant k}, S_F^A \times S_F^B, \Delta \rangle,$$

where the transition relation Δ is defined as follows:

$$\Delta\big(\overline{f}, (a_1, b_1) \dots (a_k, b_k)\big) = \big(\Delta^A(\overline{f}, a_1, \dots, a_k), \Delta^B(\overline{f}, b_1, \dots, b_k)\big).$$

From the bijectivity of σ it follows that $\mathcal{L}(C) = \mathcal{L}(A) \cap \mathcal{L}(B) = \sigma(L_1) \cap \sigma(L_2) = \sigma(L_1 \cap L_2)$, which means $L_1 \cap L_2 \in \mathrm{REG}_\sigma$. □

Theorem 6. The class of languages REG_σ with arbitrary convolution σ is closed under union.

Proof. This statement directly follows from Theorems 4 and 5 and the De Morgan's law applied to the sets L_1 and L_2: $L_1 \cup L_2 = (L_1^c \cap L_2^c)^c$. □

3.1.3 Decidability of Emptiness and Term Membership

Next, we will transfer the deciding procedures for classical tree automata to convolutional regular languages.

Theorem 7. Let σ be an arbitrary convolution and $X \in \text{REG}_\sigma$. Then the problem of checking the emptiness of X is decidable.

Proof. Let A be a tree automaton such that $\mathcal{L}(A) = \sigma(X)$, where $X = \varnothing$ if and only if $\mathcal{L}(A) = \varnothing$. The emptiness of the language of a classical tree automaton can be checked by a procedure described in [33, Theorem 1.7.4], which runs in linear time with respect to the size of the automaton. □

Theorem 8. Let σ be an arbitrary convolution and $X \in \text{REG}_\sigma$. The problem of membership of a tuple of closed terms in the set X is decidable.

Proof. Consider a tuple of closed terms $\bar{t}$ and a tree automaton A such that $\mathcal{L}(A) = \sigma(X)$. Then the following holds:

$$\bar{t} \in X \Leftrightarrow \sigma(\bar{t}) \in \sigma(X) = \mathcal{L}(A).$$

Therefore, the desired procedure consists of computing σ on the tuple $\bar{t}$ and checking whether the result belongs to the language of the automaton A using the procedure described in [33, Theorem 1.7.2]. □

3.2 Invariant Inference via Declarative Description of the Invariant-Defining Automaton

In this section, a procedure Δ which builds a first-order declarative description of the synchronous regular invariant of a CHC system is proposed. The formula $\Delta(\mathcal{P})$ has a finite model if and only if the original system $\mathcal{P}$ has an inductive invariant in the $\text{REG}_\times$ class. This gives a following method for inferring synchronous regular invariants: apply the Δ procedure to the CHC system and then apply any finite model finder to the result. To define the Δ procedure, we first introduce a language semantics for FOL, which allows one to talk about the formal languages of formulas built from formal languages of predicates.

3.2.1 Language Semantics for First-Order Logic

The formula $\Phi = \forall x_1 \ldots \forall x_n . \varphi(x_1, \ldots, x_n)$ is in *Skolem normal form (SNF)* if φ is a quantifier-free formula with free variables $x_1, \ldots, x_n$. It is known that any formula in first-order logic can be transformed to an equisatisfiable SNF formula using Skolemization procedure.

Definition 14. A tuple of terms $\langle t_1, \ldots, t_k \rangle$ is called a (n, k)-*pattern* if each of its elements t_i depends on no more than n variables from the common set of variables of this tuple.

Definition 15. A pattern is called *linear* if each variable appears in no more than one term of the tuple, and any term contains a variable no more than once. Otherwise, the pattern will be referred to as *nonlinear*.

Definition 16. By *substitution* of closed terms $u = \langle u_1, \ldots, u_n \rangle$ into a (n, k)-pattern $t = \langle t_1, \ldots, t_k \rangle$ we call a tuple of closed terms obtained by substituting terms u_i in place of variables x_i for $i = 1, \ldots, n$

$$t[u] = \langle t_1\{x_1 \leftarrow u_1, \ldots, x_n \leftarrow u_n\}, \ldots, t_k\{x_1 \leftarrow u_1, \ldots, x_n \leftarrow u_n\} \rangle.$$

Definition 17. The *downward quotient* of a language L with respect to a (n, k)-pattern t is defined as the n-ary language $L/t \triangleq \{u \in \mathcal{T}(\Sigma_F)^n \mid t[u] \in L\}$.

Example 9. Consider the Peano integer signature, which includes two functional symbols, Z and S, with arities of 0 and 1, respectively.

The tuple $\langle x_1, S(x_2), Z \rangle$ is a linear (2,3)-pattern.

The tuple $\langle S(x_1), x_1 \rangle$ is a nonlinear (1,2)-pattern.

The substitution of a term tuple $u = \langle Z,\ S(Z) \rangle$ into a (2,3)-pattern $t = \langle S(x_1),\ S(S(x_2),\ Z) \rangle$ is a tuple $t[u] = \langle S(Z),\ S(S(S(Z))), Z \rangle$.

Definition 18. Let each uninterpreted predicate symbol p correspond to a language of term tuples, denoted as $L[\![p]\!]$. The language of equality is defined as $L[\![=]\!] = \{(x,x) \mid x \in \mathcal{T}(\Sigma_F)\}$. The *language semantics of a formula in SNF*, $\forall x_1 \ldots \forall x_n . \varphi(x_1, \ldots, x_n)$ is a language $L[\![\varphi]\!]$ defined inductively as follows:

$$\begin{aligned} L[\![p(\bar{t})]\!] &\triangleq L[\![p]\!]/\bar{t} \\ L[\![\neg\psi]\!] &\triangleq \mathcal{T}(\Sigma_F)^n \setminus L[\![\psi]\!] \\ L[\![\psi_1 \wedge \psi_2]\!] &\triangleq L[\![\psi_1]\!] \cap L[\![\psi_2]\!] \\ L[\![\psi_1 \vee \psi_2]\!] &\triangleq L[\![\psi_1]\!] \cup L[\![\psi_2]\!] \end{aligned}$$

Definition 19. The formula in SNF $\Phi = \forall x_1 \ldots \forall x_n . \varphi(x_1, \ldots, x_n)$ is *satisfiable in language semantics* ($L \models \Phi$) if $L[\![\neg\varphi]\!] = \varnothing$.

Theorem 9. A formula in SNF is satisfiable in language semantics if and only if it is satisfiable in Tarski's semantics.

Proof. Set $\Phi = \forall x_1 \ldots \forall x_n . \varphi(x_1, \ldots, x_n)$. According to Herbrand's theorem, the formula Φ is satisfiable in Tarski's semantics if and only if it has a Herbrand model $\mathcal{H}$. Let $L[\![p]\!] = \mathcal{H}[\![p]\!]$. A proof is now by induction on the structure of the formula:

$$\begin{aligned} \mathcal{H} \models p(\bar{t}) &\Leftrightarrow \text{for all } \overline{u}, \bar{t}[\overline{u}] \in \mathcal{H}[\![p]\!] && \Leftrightarrow \text{for all } \overline{u}, \bar{t}[\overline{u}] \in L[\![p]\!] \\ &\Leftrightarrow \text{for all } \overline{u}, \overline{u} \in L[\![p]\!]/\bar{t} && \Leftrightarrow \text{for all } \overline{u}, \overline{u} \in L[\![p(\bar{t})]\!] \\ &\Leftrightarrow L[\![\neg p(\bar{t})]\!] = \varnothing && \Leftrightarrow L \models p(\bar{t}) \end{aligned}$$

$$\begin{aligned} \mathcal{H} \models \neg\psi &\Leftrightarrow \text{for all } \overline{u}, \mathcal{H} \not\models \psi(\overline{u}) && \Leftrightarrow \text{for all } \overline{u}, L \not\models \psi(\overline{u}) \\ &\Leftrightarrow \text{for all } \overline{u}, L[\![\neg\psi(\overline{u})]\!] \neq \varnothing && \Leftrightarrow L[\![\neg\psi]\!] = \mathcal{T}(\Sigma_F)^n \\ &\Leftrightarrow L[\![\neg\neg\psi]\!] = \varnothing && \Leftrightarrow L \models \neg\psi \end{aligned}$$

$$\begin{aligned} \mathcal{H} \models \psi_1 \wedge \psi_2 &\Leftrightarrow \mathcal{H} \models \psi_1 \text{ and } \mathcal{H} \models \psi_2 && \Leftrightarrow L \models \psi_1 \text{ and } L \models \psi_2 \\ &\Leftrightarrow L[\![\neg\psi_1]\!] = \varnothing \text{ and } L[\![\neg\psi_2]\!] = \varnothing \\ &\Leftrightarrow L[\![\psi_1]\!] = \mathcal{T}(\Sigma_F)^n \text{ and } L[\![\psi_2]\!] = \mathcal{T}(\Sigma_F)^n && \Leftrightarrow L[\![\psi_1 \wedge \psi_2]\!] = \mathcal{T}(\Sigma_F)^n \\ &\Leftrightarrow L[\![\neg(\psi_1 \wedge \psi_2)]\!] = \varnothing && \Leftrightarrow L \models \psi_1 \wedge \psi_2 \end{aligned}$$

Lastly, the De Morgan's law can be applied to prove the induction step for disjunction. □

Theorem 10. Let $L \in \text{REG}_\times$ be a language of tuples with dimension n. Then the downward quotient L/t with respect to the linear pattern $t = \langle x_1, \ldots, x_{i-1}, f(y_1, \ldots, y_m), x_{i+1}, \ldots x_n \rangle$ also belongs to the class $\text{REG}_\times$.

Proof. Without loss of generality, consider the pattern $t = \langle f(y_1, \ldots, y_m), x_2, \ldots x_n \rangle$. Let $\sigma_{fc}(L) = \mathcal{L}(A)$, where $A = \langle S, \Sigma_F^{\leqslant n}, S_F, \Delta \rangle$. Consider the automaton $A' = \langle S', \Sigma_F^{\leqslant n-1+m}, S'_F, \Delta' \rangle$, it's every state stores up to $n-1$ functional symbols and up to m^n states of automaton A, that is, $S' = \Sigma^{\leqslant n-1} \times S^{\leqslant m^n}$.

Next, we will define a tree automaton A' in such a way that the following property holds:

$$A'[\sigma_{fc}(\overline{u}, g_2(\overline{s}_2), \ldots, g_n(\overline{s}_n))] = \langle \langle g_2, \ldots, g_n \rangle, \langle A[\sigma_{fc}(t)] \mid t \in \overline{u} \times \overline{s}_2 \times \ldots \times \overline{s}_n \rangle \rangle . \tag{3.1}$$

The set of final states of the automaton A' are:

$$S'_F = \{ \langle \langle f_2, \ldots, f_n \rangle, \overline{q} \rangle \mid \Delta(\langle f, f_2, \ldots, f_n \rangle, \overline{q}) \in S_F \}.$$

Thus, by property 3.1 we have the following:

$$\begin{aligned} A'[\sigma_{fc}(\overline{u}, g_2(\overline{s}_2), \ldots, g_n(\overline{s}_n))] \in S'_F \Leftrightarrow \\ \Delta(\langle f, g_2, \ldots, g_n \rangle, \langle A[\sigma_{fc}(t)] \mid t \in \overline{u} \times \overline{s}_2 \times \ldots \times \overline{s}_n \rangle) \in S_F \Leftrightarrow \\ A[\sigma_{fc}(f(\overline{u}), g_2(\overline{s}_2), \ldots, g_n(\overline{s}_n))] \in S_F. \end{aligned}$$

To define the transition relation Δ', let us examine the unfolding of the application of A' automaton:

$$A' \left[\sigma_{fc} \left(f_1(\overline{t}_1), \ldots, f_m(\overline{t}_m), g_2(\overline{u}_2), \ldots, g_n(\overline{u}_n) \right) \right] = \Delta' \left(\langle f_1, \ldots, f_m, g_2, \ldots, g_n \rangle, \overline{a}' \right), \tag{3.2}$$

where

$$\overline{a}' = (A' \left[\sigma_{fc} \left(\overline{t}, \overline{h} \right) \right]) \mid (\overline{t}, \overline{h}) = (\overline{t}, h_2(\overline{s}_2), \ldots, h_n(\overline{s}_n)) \in (\overline{t}_1 \times \ldots \times \overline{t}_m) \times (\overline{u}_2 \times \ldots \times \overline{u}_n)) =$$

$$= (\langle \langle h_2, \ldots, h_n \rangle, (A \left[\sigma_{fc} \left(\overline{b} \right) \right] \mid \overline{b} \in \overline{t} \times \overline{s}_2 \times \ldots \times \overline{s}_n) \rangle \mid (\overline{t}, h_2(\overline{s}_2), \ldots, h_n(\overline{s}_n)) \in (\overline{t}_1 \times \ldots \times \overline{t}_m) \times (\overline{u}_2 \times \ldots \times \overline{u}_n)) .$$

In order to make automaton A' satisfy the property 3.1, the left-hand side of the equation 3.2 should also be equal to the following pair:

$$\left\langle \langle g_2, \ldots, g_n \rangle, \left(A \left[\sigma_{fc} \left(f_i(\overline{t}_i), \overline{h} \right) \right] \mid \overline{h} = (h_2(\overline{s}_2), \ldots, h_n(\overline{s}_n)) \in \overline{u}_2 \times \ldots \times \overline{u}_n \right) \right\rangle .$$

By definition, the second element of this pair is equal to the following expression:

$$(\Delta \left(\langle f_i, h_2, \ldots, h_n \rangle, (A \left[\sigma_{fc} \left(\overline{b} \right) \right] \mid \overline{b} \in \overline{t}_i \times \overline{s}_2 \times \ldots \times \overline{s}_n) \right) \mid (h_2(\overline{s}_2), \ldots, h_n(\overline{s}_n)) \in \overline{u}_2 \times \ldots \times \overline{u}_n) .$$

In the last expression, each element $A \left[\sigma_{fc} \left(\overline{b} \right) \right]$ is guaranteed to be present among the arguments of the transition relation Δ' (denoted as $\overline{a}'$). Therefore, based on the given equalities, a valid definition for Δ' can be built by replacing all occurrences of $A \left[\sigma_{fc} \left(\overline{b} \right) \right]$ in the last expression and $\overline{a}'$ with the free variables with state sorts.

For the automaton A' it holds that $\mathcal{L}(A) = \sigma_{fc}(L/t)$. □

Theorem 11. Let $L \in \text{REG}_\times$, and tuple t be a (k,n)-pattern. Then $L/t \in \text{REG}_\times$ holds.

Proof. The language L/t can be linearized, meaning it can be represented as the intersection of downward quotients of the language L with respect to linear patterns and languages for the equalities over certain variables. The conclusion of the theorem follows from the Theorem 10, which states the closure of the $\text{REG}_\times$ under downward quotients with respect to linear patterns, and Theorem 5, which states the closure of this class under intersections. □

3.2.2 Algorithm for Building Declarative Descriptions of Synchronous Regular Invariants

This section presents a description of the algorithm Δ, which transforms a CHC system over ADTs into a first-order logic formula over the free theory; from a finite model of this formula a synchronous regular invariant of the original CHC system can be recovered.

The algorithm starts by eliminating constraints from clauses using the algorithm presented in Section 2.2.

Next, from the Horn system $\mathcal{P}$ with predicates $\mathcal{R}$, the algorithm Δ builds a first-order formula in the signature $\Sigma' = \langle \Sigma'_S, \Sigma'_F, \Sigma'_P \rangle$, where

$$
\begin{aligned}
\Sigma'_S =& \{S, \mathcal{F}\} \\
\Sigma'_F =& \{delta_X \mid X \in \mathcal{R} \cup \mathcal{P} \vee X \text{is an atom from } \mathcal{P}\} \cup \Sigma_F \cup \{prod_n \mid n \geqslant 1\} \cup \\
& \cup \{delay_{n,m} \mid n, m \geqslant 1\} \\
\Sigma'_P =& \{Final_X \mid X \in \mathcal{R} \cup \mathcal{P} \vee X \text{is an atom from } \mathcal{P}\} \cup \{Reach_C \mid C \in \mathcal{P}\} \cup \{=\}.
\end{aligned}
$$

The sort S is introduced for automaton states and the sort $\mathcal{F}$ for ADT constructors. The functional symbols *delta* are introduced for the automaton transition relations, and the predicate symbols *Reach* and *Final* are introduced for the reachable and final states, respectively. For each predicate, atom, and clause, corresponding automata are built. The functional symbol $prod_n$ of arity $S^n \mapsto S$ allows one to build states that are tuples of other states. The functional symbol $delay_{n,m}$ of arity $\mathcal{F}^n \times S^m \mapsto S$ allows one to build states that are tuples of constructors and states. The Δ algorithm returns a conjunction of declarative descriptions of synchronous automata for each clause and for each atom, which are defined below.

Let C be a clause. By definition of satisfiability in language semantics we have $L \models C \Leftrightarrow L[\![\neg C]\!] = \varnothing$. Thus, the declarative description for the clause will be a first-order formula expressing $L[\![\neg C]\!] = \varnothing$. Let $\neg C \Leftrightarrow A_1 \wedge \ldots \wedge A_{n-1} \wedge \neg A_n$, where A_i are atomic formulas. Let for each A_i there be a declarative description of the corresponding atom automaton with symbols $\langle delta_{A_i}, Final_{A_i} \rangle$. For clause C, we define an automaton with symbols $\langle delta_C, Final_C \rangle$ using the construction from the proof of Theorem 5 on the closure of automata under intersection. The declarative description for the clause is then a conjunction of universal closures over all free variables of the following four formulas:

$$
\begin{aligned}
&Final_C(q) \leftrightarrow Final_{A_1}(q_1) \wedge \ldots Final_{A_{n-1}}(q_{n-1}) \wedge \neg Final_{A_n}(q_n) \\
&delta_C(x_1, \ldots, x_k, prod(q_1^1, \ldots, q_1^n), \ldots, prod(q_l^1, \ldots, q_l^n)) = \\
&\qquad = prod(delta_{A_1}(x_1, \ldots, x_k, q_1^1, \ldots, q_l^1), \ldots, delta_{A_n}(x_1, \ldots, x_k, q_1^n, \ldots, q_l^n)) \\
&Reach_C(q_1) \wedge \ldots \wedge Reach_C(q_l) \rightarrow Reach_C(delta_C(x_1, \ldots, x_k, q_1, \ldots, q_l)) \\
&Final_C(q) \wedge Reach_C(q) \rightarrow \bot
\end{aligned}
$$

Here all x have the sort $\mathcal{F}$, and all q have the sort S. The upper indices j of the state variables q_i^j correspond to the ordinal number of the automaton of atom A_j, in which the state variable is used. The first two formulas encode the automata product construction from Theorem 5. The third formula defines the set of states reachable by the clause automaton. The last formula encodes the emptiness of the clause language ("there is no state that is both final and reachable").

Declarative description of the automaton for the atom is described in the proofs of Theorems 10 and 11. Tuples of the form $\langle \langle f_2, \ldots, f_n \rangle, \overline{q} \rangle$, where f has the sort $\mathcal{F}$ and q has the sort S, are encoded using *delay* functional symbols.

3.2.3 Correctness and Completeness

Theorem 12. The system $\Delta(\mathcal{P})$ has a finite model if and only if the Horn clause system $\mathcal{P}$ has a synchronous regular invariant with full convolution.

Proof. The proof follows from the construction of $\Delta(\mathcal{P})$, the theorems on closure under Boolean operations and complement (Theorems 4, 5, 11), Theorem 9 on satisfiability of SNF formulas in the language semantics, and the fact that any system of Horn clauses can be reduced to SNF by variable renaming. □

3.2.4 Example

Let us trace the proposed transformation Δ on the following example with an lt uninterpreted predicate symbol, which represents the strict ordering relation on Peano integers:

$$\top \to lt(Z, S(x)), \tag{C1}$$

$$lt(x, y) \to lt(S(x), S(y)), \tag{C2}$$

$$lt(x, y) \wedge lt(y, x) \to \bot \tag{C3}$$

The clause $C1$ is equivalent to the atomic formula $A_1 = lt(Z, S(x))$. For the automaton of the atomic formula A_1 $\Delta(\mathcal{P})$ will build the universal closures of the following formulas, based on the automaton for the predicate symbol lt:

$$\begin{aligned} delta_{A_1}(Z) &= delta_{lt}(Z) \\ delta_{A_1}(S, q) &= delta_{lt}(S, q) \\ Final_{A_1}(q) &\leftrightarrow Final_{lt}(delta_{lt}(Z, S, q)). \end{aligned}$$

For the clause $C1$ $\Delta(\mathcal{P})$ will build the automaton $(\delta_{C1}, Final_{C1})$ with the following formulas based on the automaton for the atomic formula A_1:

$$\begin{aligned} delta_{C1}(f,q) &= delta_{A_1}(f, q) \\ Final_{C1}(q) &\leftrightarrow \neg Final_{A_1}(q). \end{aligned}$$

Moreover, the following conditions describing the emptiness of the automaton language for the clause $C1$ and guaranteeing its satisfaction will be included in $\Delta(\mathcal{P})$.

$$\begin{aligned} Reach_{C1}(q) &\to Reach_{C1}(delta_{C1}(f, q)) \\ Reach_{C1}(q) \wedge Final_{C1}(q)) &\to \bot \end{aligned}$$

The clause $C2$ consists of two atomic formulas: $A_2 = lt(x, y)$ and $A_3 = lt(S(x), S(y))$. The automaton for the atomic formula A_2 coincides with the automaton for the predicate symbol lt. For the atomic formula A_3, we will add to $\Delta(\mathcal{P})$ the automaton $(\delta_{A_3}, Final_{A_3})$ built based on the automaton for the predicate symbol lt:

$$\begin{aligned} delta_{A_3}(f,g,q) &= delta_{lt} \\ Final_{A_3}(q) &\leftrightarrow Final_{lt}(delta_{lt}(S, S, q)). \end{aligned}$$

For the clause $C2$, we add to $\Delta(\mathcal{P})$ the automaton $(delta_{C2}, Final_{C2})$, which is built based on the automaton for the predicate symbol lt and the automaton for the atomic formula A_3:

$$delta_{C2}(f,g,q) = prod_2(delta_{lt}(f,g,q), delta_{A_3}(f,g,q))$$
$$Final_{C2}(prod_2(q_1,q_2)) \leftrightarrow Final_{lt}(q_1) \wedge \neg Final_{A_3}(q_2).$$

We add to $\Delta(\mathcal{P})$ the conditions for the emptiness of the language of the automaton ($delta_{C2}$, $Final_{C2}$) in order to guarantee the satisfaction of clause $C2$.

$$Reach_{C2}(q) \rightarrow Reach_{C2}(delta_{C2}(f,g,q))$$
$$Reach_{C2}(q) \wedge Final_{C2}(q)) \rightarrow \bot$$

The clause $C3$ consists of two atomic formulas $A4 = lt(x,y)$ and $A5 = lt(y,x)$. The automaton for the atomic formula $A4$ coincides with the automaton for the predicate symbol lt. The automaton for the atomic formula $A5$ differs from the automaton for the predicate symbol lt only in the order of the arguments. After taking the remainder by such a linear template, an automaton identical to lt is obtained.

For $C3$, we add to $\Delta(\mathcal{P})$ an automaton $(delta_{C3}, Final_{C3})$ built based on the automaton for the predicate symbol lt:

$$delta_{C3}(f,g,prod_2(q_1,q_2)) = prod_2(delta_{lt}(f,g,q_1), delta_{A_2}(g,f,q_2))$$
$$Final_{C2}(prod_2(q_1,q_2)) \leftrightarrow Final_{lt}(q_1) \wedge Final_{lt}(q_2).$$

We add to $\Delta(\mathcal{P})$ the conditions for the emptiness of the language of $(delta_{C3}, Final_{C3})$ to ensure the satisfaction of $C3$:

$$Reach_{C3}(q) \rightarrow Reach_{C3}(delta_{C3}(f,g,q))$$
$$Reach_{C3}(q) \wedge Final_{C3}(q) \rightarrow \bot$$

By running a finite-model finder on the formula $\Delta(\mathcal{P})$, from interpretations of $delta_{lt}$ and $Final_{lt}$ a synchronous regular invariant of the original Horn clause system can be extracted. The obtained invariant is based on the automaton $A_{lt} = \langle\{0,1,2\}, \Sigma_F, \{1\}, \Delta\rangle$, where for $q \in 1,2$:

$$\Delta = \begin{cases} Z & \mapsto 0 \\ \langle Z, S\rangle(0) & \mapsto 1 \\ \langle S, Z\rangle(0) & \mapsto 2 \\ \langle S, S\rangle(q) & \mapsto q \end{cases}$$

The language of this automaton is the set of pairs of Peano integers, where the first number is strictly less than the second one.

3.3 Conclusion

The considered class of synchronous regular invariants with full convolution includes regular invariants as well as a large class of classical symbolic invariants. Since a *full* convolution is used, any operations with such automata will lead to an exponential complexity "explosion". That is, it should be noted that although the proposed method can theoretically infer such invariants automatically, its effectiveness needs to be tested in practice, which is done in Chapter 6. Thus, as the proposed extension of regular languages towards elementary ones does not seem to be practical it might be more fruitful to extend elementary languages towards regular ones, e. g., by extending the signature of the constraint language with predicates for term membership in a (non-synchronous) regular language. Such class of inductive invariants and the method for invariant inference in the class are proposed in the next chapter.

Chapter 4. Collaborative Inference of Combined Invariants

This chapter proposes a method for the combined invariant inference. Combined invariants (Section 4.2.1) are invariants expressible in the extension of the constraint language with predicates checking term membership in a set from some fixed class. The method presented in Section 4.2 is an extension of a counterexample-guided abstraction refinement (CEGAR) [49] algorithm for inference of combined invariants. The modification involves a collaborative information exchange with the invariant inference algorithm for the class with which it is combined (the core idea is described in Section 4.1). The chapter is based on [36].

4.1 Core Idea of Collaborative Inference

For simplicity, the key idea of collaborative inference is presented as a modification of the CEGAR approach *for transition systems*.

4.1.1 CEGAR for Transition Systems

Let $\langle \mathcal{S}, \subseteq, 0, 1, \cap, \cup, \neg \rangle$ be a complete Boolean lattice representing sets of concrete program states.

Definition 20. A *transition system (program)* is a triple $TS = \langle \mathcal{S}, Init, T \rangle$, where $Init \in \mathcal{S}$ are the initial states, and a *transition function* $T : \mathcal{S} \mapsto \mathcal{S}$ is a function, which has the following properties:

- T is monotonous, i. e., $s_1 \subseteq s_2$ implies $T(s_1) \subseteq T(s_2)$;
- T is additive, i. e., $T(s_1 \cup s_2) = T(s_1) \cup T(s_2)$;
- $T(0) = Init$.

Definition 21. States $s \in \mathcal{S}$ are said to be *reachable* from states $s' \in \mathcal{S}$ if there exists $n \geqslant 0$ such that $s = T^n(s')$.

Definition 22. A *safety problem* is a pair of a program TS and some property $Prop \in \mathcal{S}$. A program is called *safe* with respect to this property if $T^n(Init) \subseteq Prop$ is satisfied for all n, otherwise it is called *unsafe*.

Safety is witnessed by the *(safe) inductive invariant* $I \in \mathcal{S}$, for which the following must hold:

$$Init \subseteq I, \quad T(I) \subseteq I, \quad I \subseteq Prop.$$

Since all inductive invariants are fixed points of the transition function T by definition, fixed points have the most attention in the context of searching for an inductive invariant.

Theorem 13 (cf. [4]). A program is safe if and only if it has a safe inductive invariant.

In order to *automatically infer* inductive invariants, it is common to fix some *class of invariants* $\mathcal{I} \subseteq \mathcal{S}$. A *verifier* is an algorithm which for a safety problem returns either a safe inductive invariant in the invariant class $\mathcal{I}$ if the program is safe, or a counterexample otherwise. $\mathcal{I}$ is called the *domain* of the verifier. Note that in general a verifier may not terminate, for example, in the case when the program is safe, but there is no inductive invariant in its domain that proves the safety.

Definition 23. Let there be a complete lattice $\mathcal{A} = \langle A, \sqsubseteq, \bot_{\mathcal{A}}, \top_{\mathcal{A}}, \sqcap, \sqcup \rangle$, which will be called an *abstract domain* and its elements will be called *abstract states*.

A *Galois connection* [98] or an *abstraction* is a pair of mappings $\langle \boldsymbol{\alpha}, \boldsymbol{\gamma} \rangle$ between posets $\langle \mathcal{S}, \subseteq \rangle$ and $\langle \mathcal{A}, \sqsubseteq \rangle$ such that:

$$\boldsymbol{\alpha} : \mathcal{S} \mapsto \mathcal{A} \qquad \boldsymbol{\gamma} : \mathcal{A} \mapsto \mathcal{S}$$
$$\forall x \in \mathcal{S}\ \forall y \in \mathcal{A}\ \boldsymbol{\alpha}(x) \sqsubseteq y \Leftrightarrow x \subseteq \boldsymbol{\gamma}(y)\,.$$

An abstract domain together with a Galois connection uniquely defines the class of invariants $\{\boldsymbol{\gamma}(a) \mid a \in \mathcal{A}\}$, which will also be denoted as $\mathcal{A}$. In what follows, it is assumed that checks of the form $\boldsymbol{\gamma}(a) \subseteq \mathit{Prop}$ are computable.

Definition 24. An *abstract transition function* $\hat{T} : \mathcal{A} \mapsto \mathcal{A}$ "lifts" a transition function to the abstract domain, i. e., for all $a \in \mathcal{A}$ holds:

$$\boldsymbol{\alpha}(T(\boldsymbol{\gamma}(a))) \sqsubseteq \hat{T}(a)\,.$$

The following classical theorem from [65] shows how abstractions can be applied for verification.

Theorem 14. Let $TS = \langle \mathcal{S}, \mathit{Init}, T \rangle$ be the program and Prop be the property. Then $\boldsymbol{\gamma}(a)$ is an inductive invariant of $\langle TS, \mathit{Prop} \rangle$ if there exists an element $a \in \mathcal{A}$ such that:

$$\boldsymbol{\alpha}(\mathit{Init}) \sqsubseteq a, \quad \hat{T}(a) \sqsubseteq a, \quad \boldsymbol{\gamma}(a) \subseteq \mathit{Prop}.$$

```
Input: program TS and property Prop.
Output: SAFE and inductive invariant
  or UNSAFE and counterexample.
1 ⟨α, γ⟩ ← InitialAbstraction()
2 while true
3     cex, A ← ModelCheck(TS, Prop, ⟨α, γ⟩)
4     if cex is empty then
5         return SAFE(A)
6     if IsFeasible(cex) then
7         return UNSAFE(cex)
8     ⟨α, γ⟩ ← Refine(⟨α, γ⟩, cex)
```

Listing 4.1 – CEGAR for transition systems

The pseudocode of the CEGAR approach for transition systems is shown in Listing 4.1. The algorithm starts by building an initial abstraction $\langle \alpha, \gamma \rangle$, for example, using the trivial mappings $\alpha(s) = \bot_{\mathcal{A}}$ and $\gamma(a) = 1$. Then the ModelCheck procedure uses the abstraction to build a finite sequence of abstract states $\overline{a} = \langle a_0, \ldots, a_n \rangle$ such that:

$$a_0 = \alpha(Init) \quad \text{and} \quad a_{i+1} = a_i \sqcup \hat{T}(a_i) \quad \forall i \in \{0, \ldots, n-1\}. \tag{4.1}$$

If for some i we have $\gamma(a_i) \not\subseteq Prop$, then a so-called *abstract counterexample* cex is returned, either paired with $A = 0$ (if $i = 0$), or with $A = \gamma(a_{i-1})$ which satisfies $\gamma(a_{i-1}) \subseteq Prop$. If $\gamma(a_i) \subseteq Prop$ holds for all i, and $\hat{T}(a_n) \sqsubseteq a_n$ holds at some step, then $\gamma(a_n)$ is an inductive invariant, and therefore ModelCheck returns an empty cex and $A = \gamma(a_n)$. The concept of an abstract counterexample is defined by each concrete CEGAR implementation. Yet the value returned from the ModelCheck procedure must satisfy the following property:

$$A = 0 \quad \text{or} \quad Init \subseteq A \subseteq Prop. \tag{4.2}$$

If ModelCheck returns an empty abstract counterexample, then the program is safe and CEGAR returns $\gamma(a_n)$ as an inductive invariant. Otherwise, it must be checked whether the abstract counterexample corresponds to any concrete counterexample in the source program (by a IsFeasible procedure). If so, then CEGAR

stops and returns this counterexample, otherwise it proceeds by iteratively refining the $\langle\alpha,\gamma\rangle$ abstraction to eliminate the *cex* counterexample (the REFINE procedure).

4.1.2 Collaborative Inference via CEGAR Modification

In this section, we propose an approach to the collaborative inference of combined invariants. The approach is based on the collaboration of two invariant inference algorithms and is asymmetric in the following sense. First, one of the algorithms is required to be an instance of CEGAR, while the other can be arbitrary. Secondly, the main CEGAR loop controls the entire process, repeatedly calling the second algorithm.

The proposed approach is called CEGAR($\mathcal{O}$), since the "collaboration" process can be viewed as the CEGAR algorithm calling some oracle $\mathcal{O}$. Let the classes $\mathcal{A}$ and $\mathcal{B}$ be the verifier domains of CEGAR and $\mathcal{O}$, respectively. CEGAR($\mathcal{O}$) can infer inductive invariants in the union of these classes.

Definition 25. For the $\mathcal{A} \subseteq \mathcal{S}$ and $\mathcal{B} \subseteq \mathcal{S}$ state classes, a *combined class* is defined as follows:

$$\mathcal{A} \uplus \mathcal{B} \triangleq \{A \cup B \mid A \in \mathcal{A}, B \in \mathcal{B}\}.$$

A *combined (inductive) invariant* over $\mathcal{A}$ and $\mathcal{B}$ is the inductive invariant in the class $\mathcal{A} \uplus \mathcal{B}$.

More programs can be verified with combined invariants than with verifiers for combined abstract domains alone. The collaborative approach combines the strengths of verifiers for individual classes and can converge on many problems where individual verifiers would not terminate.

Example 10 ($ForkJoin$). Consider a program that transforms parallel programs in the following way. At any step, the transformation can non-deterministically eliminate all thread operations by joining all threads with the main one ($Join$) and proceed to sequential execution (Seq). If the program ends with sequential code, then the transformation inserts forking of new threads ($Fork$), followed by arbitrary transformations of the threads. If in some fragment of the given program there is a union of threads after the generation of new ones, then this fragment does not change.

This transformation can be represented as a functional program. It does not require programming language constructs other than those associated with threads,

so the following algebraic data type can be used to represent target programs:

$$Prog ::= Seq \,|\, Fork(Prog) |\, Join(Prog).$$

For example, the term `Fork(Join(Seq))` represents a program that forks new threads, then joins them at some point, and then runs sequentially only.

The described transformation has a property that if the source program consists of a sequence of consecutive forks and joins of threads, then it can never be transformed into itself. This property together with the transformation itself can be represented by the functional program in Listing 4.2.

```
1  type Prog = Seq | Fork of Prog | Join of Prog
2  fun randomTransform() : Prog
3  fun nondet() : bool
4
5  fun tr(p : Prog) : Prog =
6      match nondet(), p with
7      | false, Seq -> Fork(randomTransform())
8      | false, Fork(Join(p')) -> Fork(Join(tr(p')))
9      | _ -> Join(Seq)
10
11 fun ok(p : Prog) : bool =
12     match p with
13     | Seq -> true
14     | Fork(Join(p')) -> ok(p')
15     | _ -> false
16
17 (* for any program p : Prog *)
18 assert (not ok(p) or tr(p) <> p)
```

Listing 4.2 – Example of a functional program with algebraic data types

The `tr` function performs the transformation on the representation of the program by the *Prog* algebraic data type, in particular, it introduces arbitrary thread transformations by calling the `randomTransform` function. The `ok` function checks that the program is a sequence of consecutive forks (`Fork`) and joins (`Join`) of threads. The statement at the end encodes the property to be checked.

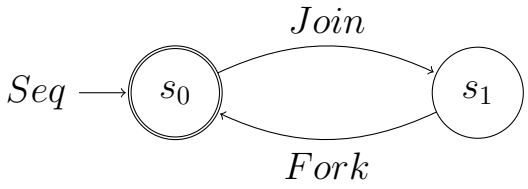
This program is safe with respect to the property but it has no inductive invariants expressible in the ELEM or REG classes. However, it has combined invariants in ELEM $\uplus$ REG. That is, for any programs $p, t : Prog$ function $\texttt{ok}(p)$ returns $\texttt{true}$ iff the following holds:

$$p \in \mathcal{E}. \tag{4.3}$$

If $\texttt{tr}(p) = t$ then the following formula is an inductive invariant for $\texttt{tr}$:

$$\neg(p = t) \vee t \notin \mathcal{E}, \tag{4.4}$$

where $t \notin \mathcal{E}$ means that the ADT term t *is not* contained in the language $\mathcal{E}$ of the following tree automaton.



Parameters: verifier $\mathcal{O}$ over domain $\mathcal{B}$
Input: a program TS and a property $Prop$
Output: $SAFE$ with a combined invariant in $\mathcal{A} \uplus \mathcal{B}$
or $UNSAFE$ with counterexample cex

```
1  ⟨α, γ⟩ ← INITIAL()
2  A ← 0
3  while true
4      async call COLLABORATE(TS, Prop, ⟨α, γ⟩, A)
5      cex, A ← MODELCHECK(TS, Prop, ⟨α, γ⟩)
6      if cex is empty the
7          return SAFE(A)
8      if ISFEASIBLE(cex) the
9          return UNSAFE(cex)
10     ⟨α, γ⟩ ← REFINE(⟨α, γ⟩, cex)
```

Listing 4.3 – Main loop of the CEGAR($\mathcal{O}$) algorithm

Description of CEGAR($\mathcal{O}$). The proposed approach is shown in Listing 4.3. The algorithm works similarly to the classic CEGAR presented in Section 4.1.1, but it also asynchronously polls the collaborating verifier $\mathcal{O}$ by calling the COLLABORATE procedure (line 4) at the beginning of each iteration. The calls are made asynchronously to prevent the algorithm from diverging if $\mathcal{O}$ diverges.

Parameters: Verifier $\mathcal{O}$ over domain $\mathcal{B}$
Input: Program $TS = \langle \mathcal{S}, Init, T \rangle$, property $Prop$, abstraction $\langle \alpha, \gamma \rangle$, state set A such that $A = 0$ or $Init \subseteq A \subseteq Prop$

1 $TS' \leftarrow \langle \mathcal{S}, T(A) \setminus A, \lambda B.\,(T(B) \setminus A) \rangle$
2 $B, cex \leftarrow \mathcal{O}(TS', Prop)$
3 **if** *cex is empty* **the**
4 $\quad$ **halt** $SAFE(A \cup B)$
5 $\widehat{cex} \leftarrow$ RECOVERCEX $(TS, Prop, \langle \alpha, \gamma \rangle, A, cex)$
6 $\langle \alpha, \gamma \rangle \leftarrow$ REFINE$(\langle \alpha, \gamma \rangle, \widehat{cex})$

Listing 4.4 – The COLLABORATE subroutine.

The **COLLABORATE** procedure is shown in Listing 4.4. Given the original safety problem, the current abstraction, and the set of states $A = \gamma(a)$ for some $a \in \mathcal{A}$, it constructs a new *residual* transition system:

$$TS' = \langle \mathcal{S}, Init', T' \rangle = \langle \mathcal{S}, T(A) \setminus A, \lambda B.\,(T(B) \setminus A) \rangle .$$

The safety of the residual system is then verified by the $\mathcal{O}$. In the Listing, $A \setminus B$ is shorthand for $A \cap \neg B$. The COLLABORATE procedure overwrites the abstraction used in CEGAR (p. 6): the $\langle \alpha, \gamma \rangle$ abstraction is global and is shared between the two procedures.

The residual system is structured as follows. Its states in the original system are reachable from states that violate the inductiveness of A. In particular, its initial states $Init'$ are $T(A) \setminus A$, i. e., the image of non-inductive states. The states $T'(Init') = T(T(A) \setminus A) \setminus A$ are reachable in one step in the residual system is the T-image of non-inductive states.

The main idea of the CEGAR($\mathcal{O}$) algorithm is to use an additional verifier $\mathcal{O}$ to *weaken* the non-inductive state set A to some fixed point in the combined class. If the second verifier finds an inductive invariant of the residual system, i. e., some inductive approximation B of non-inductive states, then $A \cup B$ will be an

inductive invariant of the original system. In other words, by building a residual system, the algorithm takes the safe but non-inductive part of the current invariant candidate and passes it to a collaborating verifier to complete it to a fixed point, i. e., inductive invariant.

Modern approaches to the inductive invariant inference, such as IC3/PDR (which can be thought of as a sophisticated version of CEGAR), monotonously *strengthen* an invariant candidate A until it becomes inductive. Because of this, the problem with such approaches is the choice of strengthening strategy [99]: due to too sharp strengthening, necessary fixed points may be missed, while due to slow strengthening, the algorithm may converge to a fixed point too slowly or even diverge.

As the proposed approach is non-monotonic, it can infer inductive invariants that cannot be inferred by verifiers run alone. In addition, it (heuristically) can speed up the invariant inference even if one of the verifiers can infer it on its own (this hypothesis is tested in Section 6.4.2). The probability of missing fixed points due to over-strengthening is reduced: even if the first verifier over-strengthens an invariant candidate, the second verifier can still detect a weaker fixed point.

Thus, if the second verifier $\mathcal{O}$ stops and returns the inductive invariant B, then Collaborate returns the combined invariant $A \cup B$. If $\mathcal{O}$ returns a concrete counterexample cex to the residual system, then Collaborate builds an abstract counterexample $\widehat{cex}$ to the original system from it and then acts like a classical CEGAR, refining the domain with $\widehat{cex}$. Note that the sets of states A and B themselves are not enough to prove the safety of the original transition system and only their union is a correct inductive invariant. In other words, collaboration is done by delegating more *simple* problems to the $\mathcal{O}$ verifier, the solution of which gives only *part* of the answer to the original problem.

Lemma 5. If the procedure Collaborate $(TS, Prop, \langle \alpha, \gamma \rangle, a)$ halts with the result $SAFE(A \cup B)$ (p. 4), then $A \cup B$ is a combined invariant of the $\langle TS, Prop \rangle$ problem.

Proof. Let us prove that $A \cup B$ is an inductive invariant by showing that it satisfies all three criteria of inductive invariants from Definition 22: this set contains all initial states, it preserves the transition relation, and it is a subset of the property.

Initial states. From the invariant (4.2) of the CEGAR algorithm we have the following cases. Either $Init \subseteq A \subseteq A \cup B$, qed. Either $A = 0$, so by definition $T(0)$, $Init = T(0) \setminus 0 = T(A) \setminus A \subseteq B \subseteq A \cup B$.

Preservation of the transition relation. From soundness of the $\mathcal{O}$ verifier we know that B is an inductive invariant $(\langle \mathcal{S}, Init', T' \rangle, Prop)$, i. e., $T(A) \setminus A \subseteq B$ ($Init'$ definition) and $T(B) \setminus A \subseteq B$ (T' definition). Thus, $(T(A) \cup T(B)) \setminus A \subseteq B$, and so $T(A) \cup T(B) \subseteq A \cup B$, hence, as the function T is additive, $T(A \cup B) \subseteq A \cup B$.

Property subset We have $A \subseteq Prop$, which follows from the invariant (4.2) of the CEGAR algorithm and $B \subseteq Prop$ by the assumption that the $\mathcal{O}$ algorithm is correct. Therefore, we have $A \cup B \subseteq Prop$. □

Counterexamples to the residual system are traces that violate the inductiveness of the current candidate invariant A. A *concrete* counterexample to the safety of the residual system (*cex* on line 2 from Listing 4.4) corresponds to some *abstract* counterexample of the original system. Therefore, CEGAR($\mathcal{O}$) is parameterized by the procedure RecoverCex, which restores an abstract counterexample to the original system from a counterexample to the residual system (p. 5). The following Section 4.2 proposes such a procedure for programs represented by CHC systems and counterexamples represented by refutation trees.

The RecoverCex procedure must satisfy the following restriction.

Restriction 1. Procedure RecoverCex $(TS, Prop, \langle \alpha, \gamma \rangle, a, cex)$ returns an abstract counterexample to the transition system $\langle TS, Prop \rangle$ with respect to the abstraction $\langle \alpha, \gamma \rangle$.

Theorem 15. If verifier $\mathcal{O}$ is correct, then verifier CEGAR($\mathcal{O}$) is also correct.

Proof. The validity of this theorem follows directly from the correctness of the original CEGAR [49], Lemma 5, and the Restriction 1. □

Theorem 16. If either CEGAR or the $\mathcal{O}$ verifier terminates on system $\langle TS, Prop \rangle$, then CEGAR($\mathcal{O}$) also terminates on the system.

Proof. If the $\mathcal{O}$ verifier terminates, then the first call to Collaborate $(TS, Prop, \langle \alpha, \gamma \rangle$ also terminates, since $Init' = T(0) \setminus 0 = Init$ and $T' = \lambda B.(T(B) \setminus 0) = T$. If CEGAR terminates, then CEGAR($\mathcal{O}$) terminates as well, since the call to Collaborate is asynchronous. □

4.2 Collaborative Invariant Inference

In this section, the collaborative inference approach for CHC systems over ADTs is presented as an instantiation of the CEGAR($\mathcal{O}$) algorithm from the previous section.

The approach has two following properties. Firstly, it infers inductive invariants expressed in the extension of the first-order logic over ADTs with constraints on term membership in tree languages $\overline{x} \in L$. Secondly, the approach extends Horn solvers with queries to first-order logic solvers, e. g., saturation-based [94] and finite-model finders [88; 89].

First, let us define the representation of the class of combined invariants.

4.2.1 Combined invariants

Definition 26. For each tree language $\mathbf{L} \subseteq |\mathcal{H}|_{\sigma_1} \times \cdots \times |\mathcal{H}|_{\sigma_m}$, define a predicate membership symbol for the language "$\in \mathbf{L}$" with arity $\sigma_1 \times \cdots \times \sigma_m$. *Membership constraint* is an atomic formula with a predicate language membership symbol. Its semantics is defined by an extension of the Herbrand semantics $\mathcal{H}$ as follows: $\mathcal{H}(\in \mathbf{L}) = \mathbf{L}$. The ADT constraint language extended with such predicates is called *first order language with membership constraints*. This language defines a class of invariants denoted by ElemReg and an abstract domain of functions from $\mathcal{R}$ predicates to formulas of the language with element-wise operations.

Example 11. The functional program from Example 10 corresponds to the following Horn clause system:

$$\begin{aligned}
p = Seq &\to ok(p) \\
p' = Fork(Join(p)) \wedge ok(p) &\to ok(p') \\
p = Seq \wedge t = Fork(p') &\to tr(p,t) \\
t = Join(Seq) &\to tr(p,t) \\
p' = Fork(Join(p)) \wedge t' = Fork(Join(t)) \wedge tr(p,t) &\to tr(p',t') \\
ok(p) \wedge tr(p,p) &\to \bot
\end{aligned}$$

It is safe, but has neither Reg nor Elem invariants. However, it has a ElemReg invariant

$$ok(p) \Leftrightarrow p \in \mathcal{E}, \quad tr(p,t) \Leftrightarrow \neg(p = t) \vee t \in \overline{\mathcal{E}},$$

where $\mathcal{E}$ is a tree language of the tree automaton from Example 10, and $\overline{\mathcal{E}}$ is its complement.

4.2.2 Horn Clause Systems as Transition Systems

Define a complete Boolean lattice of concrete states $\langle \mathcal{S}, \subseteq, 0, 1, \cap, \cup, \neg \rangle$.

$$\mathcal{S} \triangleq \text{a set of all mappings from every } P \in \mathcal{R} \text{ to a subset of } |\mathcal{H}|_P$$

$$\begin{array}{ll} s_1 \subseteq s_2 \Leftrightarrow \forall P \in \mathcal{R} \quad s_1(P) \subseteq s_2(P) & s_1 \cap s_2 \triangleq \{P \mapsto s_1(P) \cap s_2(P) \mid P \in \mathcal{R}\} \\ 0 \triangleq \{P \mapsto \varnothing \mid P \in \mathcal{R}\} & s_1 \cup s_2 \triangleq \{P \mapsto s_1(P) \cup s_2(P) \mid P \in \mathcal{R}\} \\ 1 \triangleq \{P \mapsto |\mathcal{H}|_P \mid P \in \mathcal{R}\} & \neg s \triangleq \{P \mapsto |\mathcal{H}|_P \setminus s(P) \mid P \in \mathcal{R}\} \end{array}$$

The Horn clause system $\mathcal{P}$ defines the transition system $\langle \mathcal{S}, \mathit{Init}, T \rangle$:

$$\begin{array}{l} \mathit{Init} \triangleq T\,(0) \\ T\,(s)\,(P) \triangleq \left\{ \bar{t} \mid \left(B \to P(\bar{t})\right) \text{ is a closed instance of some } C \in \mathcal{P}, s \models B \right\}. \end{array}$$

Assume without loss of generality that the Horn clause system $\mathcal{P}$ has a single query predicate Q, i. e., further, we will consider only systems obtained as follows:

$$\mathcal{P}' \triangleq \mathit{rules}\,(\mathcal{P}) \cup \{\mathit{body}\,(C)\,(\overline{x}) \to Q(\overline{x}) \mid C \text{ is a query of } \mathcal{P}\} \cup \{Q(\overline{x}) \to \bot\}.$$

The clause system defines a property for the transition system as: $\mathit{Prop}(Q) \triangleq \bot$ and for each $P \in \mathcal{R}$, $\mathit{Prop}(P) \triangleq \top$.

Proposition 1. A CHC system $\mathcal{P}$ is satisfiable if the corresponding transition system $\langle \mathcal{S}, \mathit{Init}, T \rangle$ is safe with respect to Prop.

4.2.3 Generating Residual System

The Collaborate procedure starts by building the residual system

$$\langle T\,(A) \cap \neg A, \lambda B.\,(T\,(B) \cap \neg A) \rangle\,,$$

which is passed to the collaborating verifier. The ResidualCHCs procedure from Listing 4.5 constructs a system equivalent to such a residual system by transforming the original Horn clause system $\mathcal{P}$ in two steps. It takes as input the original system $\mathcal{P}$ and an elementary model a as input.

Input: Horn clause system $\mathcal{P}$, elementary model a.
Output: Residual Horn system $\mathcal{P}'$.

1 $\Phi \leftarrow \mathcal{P}$ with atoms $P(\bar{t})$ replaced by $a(P)(\bar{t}) \vee P(\bar{t})$
2 **return** $CNF(\Phi)$

Listing 4.5 – ResidualCHCs algorithm for generation of a residual CHC system.

The procedure replaces each atom $P(t_1, \ldots, t_m)$ in the heads and bodies of the Horn system with the disjunction $a(P)(t_1, \ldots, t_m) \vee P(t_1, \ldots, t_m)$ (line 1). Then, it moves the Σ-formula from the head to the body with negation and splits the clause according to the disjunction in the body, bringing it into CNF. For example, the clause

$$P(x) \wedge \varphi(x, x') \to P(x')$$

will first turn into the formula

$$(a(P)(x) \vee P(x)) \wedge \varphi(x, x') \to (a(P)(x') \vee P(x')),$$

which after transformation into CNF (p. 2) will be divided into the following clauses:

$$a(P)(x) \wedge \varphi(x, x') \wedge \neg a(P)(x') \to P(x')$$
$$P(x) \wedge \varphi(x, x') \wedge \neg a(P)(x') \to P(x').$$

Thus, as a result of transformation into CNF, we obtain a system of clauses which semantically corresponds to the residual system from the previous section.

Example 12. Consider abstract state $a(tr)(p, t) \equiv \neg(p = t) \vee t = Join(Seq)$, $a(ok)(p) \equiv p = Seq$ and the system from Example 11. The procedure ResidualCHCs will first give the formula

$$p = Seq \to \big(p = Seq \vee ok(p)\big)$$
$$p' = Fork(Join(p)) \wedge \big(p = Seq \vee ok(p)\big) \to \big(p' = Seq \vee ok(p')\big)$$
$$p = Seq \wedge t = Fork(p') \to \big(\neg(p = t) \vee t = Join(Seq) \vee tr(p, t)\big)$$
$$t = Join(Seq) \to \big(\neg(p = t) \vee t = Join(Seq) \vee tr(p, t)\big)$$
$$p' = Fork(Join(p)) \wedge t' = Fork(Join(t)) \wedge$$
$$\wedge \big(\neg(p = t) \vee t = Join(Seq) \vee tr(p, t)\big) \to \big(\neg(p' = t') \vee t' = Join(Seq) \vee tr(p', t')\big)$$
$$\big(p = Seq \vee ok(p)\big) \wedge \big(\neg(p = p) \vee p = Join(Seq) \vee tr(p, p)\big) \to \bot,$$

which can be simplified to

$$\begin{aligned} p = Fork(Join(Seq)) &\to ok(p) \\ p' = Fork(Join(p)) \wedge ok(p) &\to ok(p') \\ t = Fork(Join(Join(Seq))) &\to tr(p,t) \\ p' = Fork(Join(p)) \wedge t' = Fork(Join(t)) \wedge p' = t' \wedge tr(p,t) &\to tr(p',t') \\ \big(p = Seq \vee ok(p)\big) \wedge \big(p = Join(Seq) \vee tr(p,p)\big) &\to \bot \end{aligned}$$

4.2.4 CEGAR($\mathcal{O}$) for CHCs: Recovering Counterexamples

This section presents a procedure for building an abstract counterexample of the original system from a concrete counterexample for the residual system obtained as $\mathcal{P}' = \textsc{ResidualCHCs}(\mathcal{P}, a)$. In other words, we present the instantiation of the RecoverCex procedure from Listing 4.4.

Abstract counterexamples. An abstract counterexample of a Horn clause system is a refutation tree, some of whose leaves may be abstract states. For a formal definition, we introduce the transformation of clauses $Q(\mathcal{P}, a)$, which for each predicate $P \in \mathcal{R}$ adds new clauses $a(P)(\overline{x}) \to P(\overline{x})$ to the system $\mathcal{P}$.

Definition 27. An *abstract counterexample* of a Horn system $\mathcal{P}$ with respect to the abstract state a is the refutation tree of the Horn system $Q(\mathcal{P}, a)$.

Let T be a refutation tree of $\mathcal{P}' = \textsc{ResidualCHCs}(\mathcal{P}, a)$. Let us present a recursive procedure for building a refutation tree T' for $\mathcal{P}'' = Q(\mathcal{P}, a)$ given the tree T.

Recursion base. Let T be a leaf $\langle C, \Phi\rangle$, where $C \in \mathcal{P}'$. Since $\Phi = body(C)$ is a predicate-free formula, then C is

$$\varphi \wedge a(P_1)(\overline{x}_1) \wedge \ldots \wedge a(P_n)(\overline{x}_n) \wedge \neg a(P)(\overline{x}) \to P(\overline{x}),$$

Let us build T' as $\langle C', \Phi'\rangle$, where

$$C' \equiv \varphi \wedge P_1(\overline{x}_1) \wedge \ldots \wedge P_n(\overline{x}_n) \to P(\overline{x}) \quad \text{and} \quad \Phi' \equiv \varphi \wedge a(P_1)(\overline{x}_1) \wedge \ldots \wedge a(P_n)(\overline{x}_n),$$

with n leaf children $\langle C'_i, a(P_i)(\overline{x}_i)\rangle$, where $C'_i \equiv a(P_i)(\overline{x}_i) \to P_i(\overline{x}_i)$. The definition of the refutation tree is trivially satisfied. Note that $\mathcal{H} \models \Phi \to \Phi'$.

Recursion step. Let T be a of node $\langle C, \Phi\rangle$ with children $\langle C_1, \Phi_1\rangle, \ldots, \langle C_n, \Phi_n\rangle$, all $C_i \in rules(P_i)$ from $\mathcal{P}'$ and

$$C \equiv \varphi \wedge a(R_1)(\overline{y}_1) \wedge \ldots \wedge a(R_m)(\overline{y}_m) \wedge \neg a(R)(\overline{y}) \wedge P_1(\overline{x}_1) \wedge \ldots \wedge P_n(\overline{x}_n) \to R(\overline{y})$$
$$\Phi \equiv \varphi \wedge a(R_1)(\overline{y}_1) \wedge \ldots \wedge a(R_m)(\overline{y}_m) \wedge \neg a(R)(\overline{y}) \wedge \Phi_1(\overline{x}_1) \wedge \ldots \wedge \Phi_n(\overline{x}_n).$$

Due to the iteration of the recursion, we already have the corresponding nodes $\langle C'_1, \Phi'_1 \rangle, \ldots, \langle C'_n, \Phi'_n \rangle$, so define:

$$C' \equiv \varphi \wedge R_1(\overline{y}_1) \wedge \ldots \wedge R_m(\overline{y}_m) \wedge P_1(\overline{x}_1) \wedge \ldots \wedge P_n(\overline{x}_n) \to R(\overline{y})$$
$$\Phi' \equiv \varphi \wedge a(R_1)(\overline{y}_1) \wedge \ldots \wedge a(R_m)(\overline{y}_m) \wedge \Phi'_1(\overline{x}_1) \wedge \ldots \wedge \Phi'_n(\overline{x}_n).$$

For each predicate R_j, add children: $\left\langle C'_{n+j}, a(R_j)(\overline{y}_j) \right\rangle$, where $C'_{n+j} \equiv a(R_i)(\overline{y}_i) \to R_j(\overline{y}_j)$. For each i, $\mathcal{H} \models \Phi_i \to \Phi'_i$ by induction, so for their conjunction we have $\mathcal{H} \models \Phi \to \Phi'$.

Eventually the recursion will come to the root of the tree T, which is some vertex $\langle C, \Phi \rangle$, where C is a query from the system $\mathcal{P}'$. A tree T' with root $\langle C', \Phi' \rangle$ is recursively built for it. By induction we have $\mathcal{H} \models \Phi \to \Phi'$. Since Φ is a satisfiable Σ-formula, so is Φ'. Thus, T' is the refutation tree of the $\mathcal{P}''$ system.

Proposition 2. The procedure RecoverCex is linear in the number of nodes of the input refutation tree.

4.2.5 Instantiating Approach within IC3/PDR

The above algorithm allows inferring combined invariants in the class Elem ⊎ Reg, i. e., inductive invariants expressible by formulas of the form $\varphi(\overline{x}) \vee \overline{x} \in L$, where φ is a first-order formula over ADTs and L is a tree language. The implementation of the IC3/PDR approach as a complex instantiation of CEGAR can be generalized for the automatic invariant inference in the full quantifier-free fragment ElemReg with formulas of the following form:

$$\bigwedge_i (\varphi_i(\overline{x}) \vee \overline{x} \in L_i). \tag{4.5}$$

IC3/PDR represents its abstract state as a conjunction of formulas (called *lemmas*). In other words, in the procedure ResidualCHCs$(\mathcal{P}, a)$ (see section 4.2.3), the function a maps each uninterpreted symbol P to some conjunction $\bigwedge_i \varphi_i$. We generalize the approach by replacing each uninterpreted predicate symbol P with *disjunction of conjunctions* $\bigwedge_i (\varphi_i(\overline{t}) \vee L_i(\overline{t}))$ with fresh predicate symbols L_i. Thus, inductive invariants of the above form 4.5 will be inferred by the modified IC3/PDR.

4.3 Conclusion

The proposed class of combined invariants, built on regular invariants, allows one to express both classical symbolic invariants and complex recursive relations. Thus, the proposed class of invariants should be expressive enough for practice. Additionally, an efficient invariant inference method in the class has been proposed. The method reuses existing efficient invariant inference algorithms for combined classes by a minor modification of one of them. The next chapter is dedicated to a theoretical comparison of existing and proposed classes of inductive invariants.

Chapter 5. Theoretical Comparison of Inductive Invariant Classes

This chapter provides a theoretical comparison of existing and proposed classes of inductive invariants for programs with algebraic data types. We consider only classes for which there are fully automatic invariant inference methods: elementary invariants (Elem, inferred by Spacer [22] and HoIce [25]), elementary invariants with term size constraints (SizeElem, inferred by Eldarica [24]), regular invariants (Reg, inferred by RCHC [26], as well as the method from Chapter 2), synchronous regular invariants (Reg_+, $\text{Reg}_\times$, inferred by RCHC [26], as well as by the method from Chapter 3) and combined invariants (ElemReg, inferred by the method from Chapter 4). The chapter is partly based on [35].

The comparison is based on the key properties of invariant classes: closure with respect to Boolean operations, decidability of the term membership problem, decidability of checking an invariant for emptiness (Section 5.1), and expressive power (Section 5.2). The results of the theoretical comparison are shown in Tables 5.1 and 5.2. Section 5.3 presents an overview of representation methods for infinite sets of terms based on generalizations of tree automata that might serve as classes of inductive program invariants in the future.

5.1 Closure under Boolean Operations and Decidability

Closure and decidability properties for the investigated classes are summarized in Table 5.1. A footnote in each cell of the table refers to the theorem stating the claimed result; the absence of a footnote indicates that the fact asserted in the cell is obvious. For example, the closure of the classes Elem, SizeElem, and ElemReg with respect to Boolean operations is clear, as they are syntactically constructed as first-order languages with the corresponding operations.

5.2 Invariant Classes Expressivity

Comparison of the invariant classes expressiveness is presented in the Table 5.2. Since some classes are built as syntactic extensions of other classes (for example,

[1] *even* $\in$ Reg $\setminus$ SizeElem (Theorem 21)
[2] *lr* $\in$ Elem $\setminus$ $\text{Reg}_\times$ (Lemma 7)
[3] *lt* $\in$ $\text{Reg}_+ \setminus$ Reg (Theorem 17)
[4] *node* $\in$ $\text{Reg}_\times \setminus \text{Reg}_+$ (Lemma 6)

Table 5.1 – Theoretical comparison of inductive invariant classes

Property \ Class	Elem	SizeElem	Reg	Reg_+	$\text{Reg}_\times$	ElemReg
Closed under $\cap$	Yes	Yes	Yes[1]	Yes[2]	Yes[2]	Yes
Closed under $\cup$	Yes	Yes	Yes[1]	Yes[2]	Yes[2]	Yes
Closed under $\setminus$	Yes	Yes	Yes[1]	Yes[2]	Yes[2]	Yes
$\bar{t} \in I$ is decidable	Yes[3]	Yes[4]	Yes[5]	Yes[7]	Yes[9]	Yes[10]
$I = \varnothing$ is decidable	Yes[3]	Yes[4]	Yes[6]	Yes[8]	Yes[9]	Yes[10]
Recursive relations are expressible	No	Partially	Yes	Yes	Yes	Yes
Synchronous relations are expressible	Yes	Yes	No	Partially	Yes	Yes

[1] see [33, property 3.2.9]
[2] see Section 3.1.2
[3] see [93]
[4] see [100]
[5] see [33, Section 3.2.1 and Th. 1.7.2]
[6] see [33, Section 3.2.1 and Th. 1.7.4]
[7] see [33, Def. 3.2.1 and Th. 1.7.2]
[8] see [33, Def. 3.2.1 and Th. 1.7.4]
[9] see Section 3.1.3
[10] see [101, Corollary 2]

Table 5.2 – Theoretical comparison of inductive invariant classes expressivity

Class	Elem	SizeElem	Reg	Reg_+	$\text{Reg}_\times$	ElemReg
Elem	$\varnothing$	$\varnothing$	$lr^{1,4,5}$	$lr^{1,5}$	lr^{1}	$\varnothing$
SizeElem	∞	$\varnothing$	$lr^{1,4,5}$	$lr^{1,5}$	lr^{1}	lt^{3}
Reg	$even^{2}$	$even^{2}$	$\varnothing$	$\varnothing^{4}$	$\varnothing^{4,5}$	$\varnothing$
Reg_+	$even^{2,7}$	$even^{2,4}$	∞^{4}	$\varnothing$	$\varnothing^{5}$	lt^{3}
$\text{Reg}_\times$	$even^{2,4,5}$	$even^{2,4,5}$	$\infty^{4,5}$	∞^{5}	$\varnothing$	$lt^{3,5}$
ElemReg	∞	$even^{2}$	∞	$lr^{1,5}$	lr^{1}	$\varnothing$

[1] $lr \in \text{Elem} \setminus \text{Reg}_\times$ (Lemma 7)
[2] $even \in \text{Reg} \setminus \text{SizeElem}$ (Th. 21)
[3] see Th. 17
[4] $\text{Reg} \subseteq \text{Reg}_+$ [33, Prop. 3.2.6]
[5] $\text{Reg}_+ \subseteq \text{Reg}_\times$ [26, Th. 11]

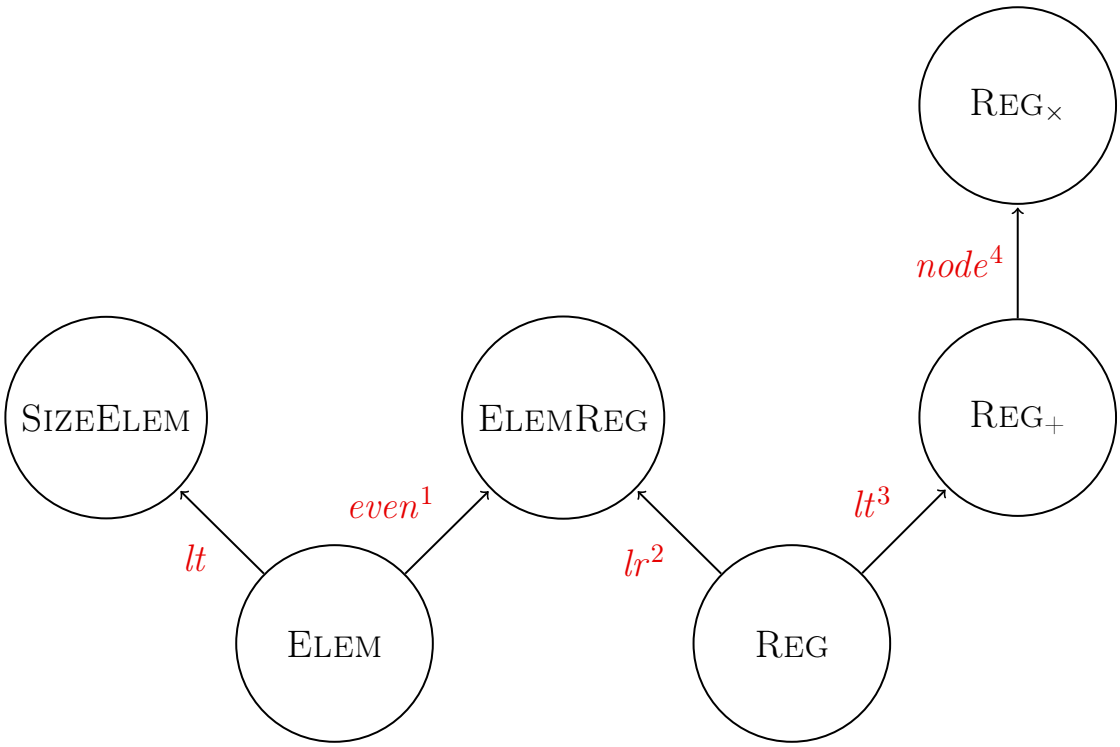


Figure 5.1 – Inclusion relations between classes of inductive invariants over ADTs.

REG$_+$ and REG$_\times$ both extend REG), and some classes are syntactically very different (for example, REG and ELEM), the relationships between the sets they represent are not obvious. It is important to distinguish invariant classes for the analysis of invariant inference algorithms, in particular, for understanding the limits of their applicability. If the invariants of problems of some kind do not lie in the class of invariants inferred by the given algorithm, then this algorithm will not terminate on problems of this kind. Therefore, these relationships are presented in the table 5.2.

For class A in the row and class B in the column, the corresponding cell contains a footnote, and either the symbol $\varnothing$, or ∞, or the name of a certain clause system from this work. Each cell should be read as an answer to the question: "What does the class $A \setminus B$ contain?" If the cell contains $\varnothing$, then $A \setminus B = \varnothing$. If the cell contains ∞, then $B \subseteq A$. Finally, if the cell contains the name $\mathcal{P}$ of some system, then the classes A and B are incomparable, i. e., $\mathcal{P} \in A \setminus B \neq \varnothing$ and $B \setminus A \neq \varnothing$. The footnote refers to the corresponding theorem presented in this thesis. The absence of a footnote indicates that the stated fact is obvious. For example, the cell SIZEELEM $\setminus$ ELEM contains ∞ without a footnote, because the SIZEELEM class is a syntactic extension of the ELEM class, and therefore includes at least the same invariants.

Figure 5.1 presents inclusion relations between invariant classes separately for convenience. An edge from class A to class B labeled $\mathcal{P}$ means that $A \subsetneq B$ and $\mathcal{P} \in B \setminus A$.

5.2.1 Inexpressivity in Synchronous Languages

Example 13 (*node*). Consider the following set of terms over the binary tree algebraic type $Tree ::= left \,|\, node(Tree, Tree)$:

$$node \triangleq \{ \langle Node(y, z), y, z\rangle \mid y, z \in \mathcal{T}(\Sigma_F)\}.$$

This example allows us to separate classes of synchronous regular invariants with full and standard convolution, as the next lemma shows.

Lemma 6. There are synchronous regular invariants with full convolution that are inexpressible using only standard convolution, i. e., $node \in \text{REG}_\times \setminus \text{REG}_+$.

Proof. $node \in \text{REG}_\times$ follows from application of Theorem 11 to a language of equality of two terms and a linear template $\langle Node(y, z), y, z\rangle$. The validity of $node \notin \text{REG}_+$ is shown in [33, Ex. 3.2] by applying the pumping lemma for tree automata languages to *node*. □

Example 14 (*lr*). Consider the following set of terms over the binary tree algebraic type $Tree ::= left \,|\, node(Tree, Tree)$:

$$lr \triangleq \{x \mid \exists t \,.\, x = node(t, t)\}.$$

This set lies in the class of elementary invariants, yet it cannot be expressed by any synchronous tree automaton even with full convolution, as the next lemma shows.

Lemma 7. There are elementary invariants which are not expressible regularly with full synchronization, i. e., $lr \notin \text{REG}_\times$.

Proof. In [33, Ex. 1.4], by application of the pumping lemma to *lr*, it is shown that $lr \notin \text{REG}$. By Lemma 4, this implies $lr \notin \text{REG}_\times$. □

5.2.2 Inexpressivity in Combined Languages

Theorem 17. The intersection of classes SizeElem and Reg_+ does not belong to the class ElemReg, i. e., $lt \in \text{SizeElem}$, $lt \in \text{Reg}_+$, $lt \notin \text{ElemReg}$.

Proof. The set lt is expressed by the following SizeElem-formula:

$$\varphi(x, y) \triangleq size(x) < size(y).$$

The fact that $lt \in \text{Reg}_+$ was shown in Example 8.

Let us now show that lt does not belong to ElemReg. Note that the algebraic type of Peano integers is isomorphic to natural numbers (with zero). Furthermore, formulas representing ElemReg are isomorphic to formulas in the signature of extended Presburger arithmetic without addition and order. Consider this signature $\Sigma = \langle \Sigma_S, \Sigma_F, \Sigma_P \rangle$, which includes a unique sort for natural numbers ($\Sigma_S = \{\mathbb{N}\}$), a constant 0, and a unique successor function symbol s, where $s(x)$ is interpreted as $x + 1$, $\Sigma_F = \{0, s\}$, as well as predicate symbols of equality and divisibility for all constants ($c \mid x$ is interpreted as x is divisible by c, $\Sigma_P = \{=\} \cup \{c \mid \_, c \in \mathbb{N}\}$). Since the set lt represents a strict order relation, in order to prove the original statement it is necessary to show that the standard order relation on natural numbers is inexpressible in the theory of the standard model $\mathcal{N}$ of signature Σ.

Let us prove this proposition by extending the proof from [102, Sec. 2] for an arithmetic with the same signature but without divisibility predicates. Consider the model $\mathcal{M} = (\mathbb{N} \cup \mathbb{N}^*, s, c \mid \_)$, where $\mathbb{N}^*$ is defined as the set of symbols $\{n^* \mid n \in \mathbb{N}\}$; $c \mid n$ and $c \mid n^*$ are true only when c divides n; and the successor function is defined as follows:

$$s(n) \triangleq n + 1$$
$$s(n^*) \triangleq (n + 1)^*$$

Model $\mathcal{M}$ is an elementary extension of the model $\mathcal{N}$, so if some formula $\psi(x, y)$ defines a linear order on $\mathcal{N}$, then it defines a linear order on $\mathcal{M}$. Note that the following mapping σ is an automorphism of model $\mathcal{M}$:

$$\sigma(n) \triangleq n^*$$
$$\sigma(n^*) \triangleq n$$

From the fact that σ is an automorphism of the model $\mathcal{M}$, it follows that for any $x, y \in \mathbb{N} \cup \mathbb{N}^*$, it is true that $\mathcal{M} \models \psi(x, y) \Leftrightarrow \mathcal{M} \models \psi(\sigma(x), \sigma(y))$.

Since the formula ψ is assumed to express a linear order, without loss of generality, assume that $\mathcal{M} \models \psi(0, 0^*)$. However, by applying the automorphism σ, we get that $\mathcal{M} \models \psi(0^*, 0)$, which contradicts the axioms of order. Therefore, no formula of a given signature can represent a linear order. It follows that *lt* does not lie in the ElemReg class. □

5.2.3 Inexpressivity in Elementary Languages

This section introduces pumping lemmas for first-order languages: the ADT constraint language and the ADT constraint language extended with term size constraints.

First pumping lemmas arose in the theory of formal languages [103] applied to finite automata and context-free grammars. In general, a pumping lemma claims that for all languages in some class (e. g., regular or context-free languages), any sufficiently large word can be "pumped". In other words, some parts of the word can be indefinitely enlarged, and the pumped word will still be a part of the language. Pumping lemmas are useful to prove that an invariant is not expressible in some class: you assume that the invariant belongs to a class, and then you apply a specialized pumping lemma for this class and get some pumped set. If the set cannot be an inductive invariant, then a contradiction has been obtained; therefore, the invariant is inexpressible in the given class.

To formally state pumping lemmas, we first define the following extension Elem$*$ of the constraint language, which admits quantifier elimination. For every ADT $\langle \sigma, C \rangle$ and every constructor $f \in C$ of arity $\sigma_1 \times \cdots \times \sigma_n \to \sigma$ for some sorts of $\sigma_1, \ldots, \sigma_n$, introduce *selectors* $g_i \in S$ of arity $\sigma \to \sigma_i$ for each $i \leqslant n$ with standard semantics given as follows: $g_i(f(t_1, \ldots, t_n)) \triangleq t_i$.

Theorem 18 (see [93]). Any Elem-formula is equivalent to some quantifier-free Elem$*$-formula.

Let us give some auxiliary definitions.

Definition 28. We define the height of a closed term inductively as follows:

$$\mathcal{H}eight(c) \triangleq 1$$
$$\mathcal{H}eight(c(t_1, \ldots, t_n)) \triangleq 1 + \max_{i=1}^{n} \big(\mathcal{H}eight(t_i)\big).$$

Let us call a *path* a (possibly empty) sequence of selectors $s \triangleq S_1 \dots S_n$, where for each i, S_i has sort $\sigma_i \to \sigma_{i-1}$. For each term t of sort σ_n, let $s\,(t) \triangleq S_1(\dots(S_n(t))\dots)$. For closed terms g, we redefine $s\,(g)$ as a computed subterm of g in s. In what follows, paths will be denoted by letters p, q, r, s.

We say that two paths p and q *overlap* if one of them is a suffix of the other. For pairwise non-overlapping paths $p_1, \dots, p_n$, by the notation $t[p_1 \leftarrow u_1, \dots, p_n \leftarrow u_n]$ we mean the term obtained by simultaneously replacing subterms $p_i\,(t)$ in t with terms u_i. For a finite sequence of pairwise distinct paths $P = (p_1, \dots, p_n)$ and some set of terms $U = (u_1, \dots, u_n)$ we redefine the notation and write $t[P \leftarrow U]$ instead of $t[p_1 \leftarrow u_1, \dots, p_n \leftarrow u_n]$, and also $t[P \leftarrow t]$ instead of $t[p_1 \leftarrow t, \dots, p_n \leftarrow t]$.

Now let us define a set of paths that will be pumped.

Definition 29. A term t is a *leaf term* of sort σ, if it is a parameterless constructor, or $t = c(t_1, \dots, t_n)$, where all t_i are leaf terms and t does not contain any proper sub-terms of sort σ. For a closed term g and sort σ we define $leaves_\sigma(g) \triangleq \{p \mid p\,(g)$ is a leaf term of sort $\sigma\}$.

Lemma 8 (Pumping Lemma for ELEM). Let $\mathbf{L}$ be an elementary language of n-tuples. Then, there exists a constant $K > 0$ satisfying:

- for every n-tuples of ground terms $\langle g_1, \dots, g_n\rangle \in \mathbf{L}$,
- for any i such that $\mathcal{H}eight(g_i) > K$,
- for all infinite sorts $\sigma \in \Sigma_S$ and
- for all paths p with a length greater than K,
- there exist finite sets of paths P_j such that $p \in P_i$,
- for all $p_1, p_2 \in \bigcup_j P_j$ it is true that $p_1(g) = p_2(g)$,
- and there is $N \geqslant 0$, such that
- for all t of sort σ with $\mathcal{H}eight(t) > N$ it holds that:

$$\langle g_1[P_1 \leftarrow t], \dots, g_i[P_i \leftarrow t], \dots, g_n[P_n \leftarrow t]\rangle \in \mathbf{L}.$$

Proof. The proof is given in [35]. □

In fact, Lemma 8 states that for sufficiently large tuples of terms one can take any of the deepest subterms, replace them with *arbitrary* terms t and *still* get a tuple of terms from the given language. This lemma formalizes the fact that a constraint language over an ADT theory can only describe equalities and disequalities between subterms of bounded depth: if you go deep enough and replace leaf terms

with arbitrary terms, then the initial and resulting terms are *indistinguishable* by the first-order formula.

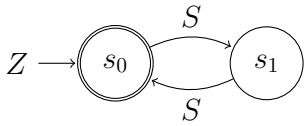
Theorem 19. There are regular but non-elementary invariants, i. e., $\textsc{Reg} \setminus \textsc{Elem} \neq \varnothing$.

Proof. Consider the Horn clause system over the algebraic type of Peano integers $Nat ::= Z \mid S\ Nat$, which checks the parity of numbers and states that no two successive numbers can be even:

$$\begin{aligned} x = Z &\to ev(x) \\ ev(y) \wedge x = S(S(y)) &\to ev(x) \\ ev(x) \wedge ev(y) \wedge x = S(y) &\to \bot \end{aligned}$$

The system in this example has a single inductive invariant — the set $E = \{S^n(Z) \mid n \geqslant 0\}$. This can be proved by contradiction: if this set is extended by some odd number $E \cup \{S^{2n+1}(Z)\} \subseteq E'$, then the query condition will be violated for $x = S^{2n}(Z)$ and $y = S^{2n+1}(Z)$. Thus, the set E is the only safe inductive invariant of this system.

It is easy to see that the set E is expressible by the following tree automaton (and hence the system has an inductive invariant in $\textsc{Reg}$):



Let us prove that the set E cannot be expressed by a constraint language formula. Assume that it is. Take the constant $K > 0$ from Lemma 8. Let $g \equiv S^{2K}(Z) \in E, \sigma = Nat, p = S^{2K}$. Further, $\bigcup_j leaves_\sigma(g_j) = leaves_\sigma(g) = \{p\}$, so $P = \{p\}$. Then, by Lemma 8, there exists $N \geqslant 0$ such that if we set $t \equiv S^{2N+1}(Z)$, then $g[P \leftarrow t] \equiv S^{2K}(S^{2N+1}(Z)) \in E$. Therefore, the set E contains an odd number, which contradicts the definition of this set of even numbers. □

Next, we introduce a similar lemma for a first-order language with term size constraints. For this purpose, consider the corresponding extension with selectors $\textsc{SizeElem}*$, which admits quantifier elimination.

Theorem 20 (see [104]). Any $\textsc{SizeElem}$-formula is equivalent to some quantifier-free $\textsc{SizeElem}*$-formula.

Definition 30. Borrowing notation from [100], we denote $\mathbb{T}^k_\sigma = \{t \text{ has sort } \sigma \mid size(t) = k\}$. For each ADT sort σ we define the set of term sizes as $\mathbb{S}_\sigma = \{size(t) \mid t \in |\mathcal{H}|_\sigma\}$. A *linear set* is a set of the form $\{\mathbf{v} + \sum_{i=1}^{n} k_i \mathbf{v_i} \mid k_i \in \mathbb{N}_0\}$, where all $\mathbf{v}$ and $\mathbf{v_i}$ are vectors over $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$.

Definition 31. An ADT sort σ is called *expanding* if for every natural number n there exists a bound $b(\sigma, n) \geqslant 0$ such that for every $b' \geqslant b(\sigma, n)$, if $\mathbb{T}^{b'}_\sigma \neq \varnothing$, then $\left|\mathbb{T}^{b'}_\sigma\right| \geqslant n$ An ADT signature is expanding if all of its sorts are expanding.

Lemma 9 (Pumping Lemma for SizeElem). Let the ADT signature be expanding and let $\mathbf{L}$ be an elementary language of n-tuples with size constraints. Then, there exists a constant $K > 0$ satisfying:

- for every n-tuple of ground terms $\langle g_1, \ldots, g_n \rangle \in \mathbf{L}$,
- for any i, such that $\mathcal{H}eight(g_i) > K$,
- for all infinite sorts $\sigma \in \Sigma_S$, and
- for all paths $p \in leaves_\sigma(g_i)$ with length greater than K,
- there exists an infinite linear set $T \subseteq \mathbb{S}_\sigma$, such that
- for all terms t of sort σ with sizes $size(t) \in T$,
- there exist sequences of paths P_j, with no path in them being a suffix of path p,
- and sequences of terms U_j, such that

$$\langle g_1[P_1 \leftarrow U_1], \ldots, g_i[p \leftarrow t, P_i \leftarrow U_i], \ldots, g_n[P_n \leftarrow U_n] \rangle \in \mathbf{L}.$$

Proof. The proof is given in [35]. □

The core idea of Lemma 9 is that having a language from the SizeElem class, a sufficiently large term g from it, and a sufficiently large path p, one can replace $p\,(g)$ by an arbitrary term t (limited only by the size, which must be in some linear infinite set T), and again get the term from the same language. This fact, in turn, means that in each infinite language from the SizeElem class there are subterms that are indistinguishable by its formulas.

Example 15 (*even*). Consider the Horn clause system over the binary tree algebraic type $Tree ::= left \,|\, node(Tree, Tree)$, which checks whether the number of

nodes in the leftmost branch of the tree is even.

$$x = leaf \to even(x)$$
$$x = node(node(x', y), z) \wedge even(x') \to even(x)$$
$$even(x) \wedge even(node(x, y)) \to \bot$$

As shown below, this system does not have an invariant that can be expressed by a first-order formula even with term size constraints.

Theorem 21. There are regular invariants that are not elementary invariants with term size constraints, i. e., $even \in \text{REG} \setminus \text{SIZEELEM}$.

Proof. The invariant *even* can be expressed by the automaton $\langle \{s_0, s_1\}, \Sigma_F, \{s_0\}, \Delta \rangle$ with the transition rules Δ defined as follows.

$$leaf \mapsto s_0$$
$$node(s_0, s_0) \mapsto s_1$$
$$node(s_0, s_1) \mapsto s_1$$
$$node(s_1, s_0) \mapsto s_0$$
$$node(s_1, s_1) \mapsto s_0$$

With the pumping lemma, we can prove that the *even* invariant does not belong to the SIZEELEM class. First, it is obvious that the sort *Tree* is expanding. Suppose *even* is in the class SIZEELEM and has an invariant $\mathbf{L}$. Take $K > 0$ from Lemma 9. Let $g \in \mathbf{L}$ be a complete binary tree of height $2K$, $\sigma = Tree, p = Left^{2K}$. Take the infinite linear set T from the lemma. We can find some $n \in T, n > 2$ and $t = node(leaf, t')$ for some t' such that $size(t) = n$. By Lemma 9 there is a sequence of paths P and a sequence of terms U, and none of the elements in P is a suffix of p; it must also be true that $g[p \leftarrow t, P \leftarrow U] \in \mathbf{L}$, so the leftmost path in the tree must have an even length. However, the leftmost path $p = Left^{2K}$ contains the term $node(leaf, t')$, so the path to the leftmost leaf of the tree is $2K - 1 + 2 = 2K + 1$, which is an odd number. So, we have a contradiction with the fact that the path to the leftmost leaf in each term of the set *even* has an even length, which means that *even* does not belong to the SIZEELEM class. □

5.3 Finite Representations of Term Sets

So far a number of various classes of invariants for Horn clause systems over ADTs, such as Elem, Reg, Reg_+, $\text{Reg}_\times$, were examined and proposed. A key requirement for classes of inductive invariants over ADTs is the ability to represent infinite sets of term tuples by finite means so that a finite computer can work with them. Moreover, these representations should provide closure and decidability of certain operations discussed in detail in this chapter. Such finite representations of term sets are also studied in other areas of computer science and can be used for invariant inference.

The problem of finite term set representation can be stated as the task of Herbrand model representation, which is addressed in the field of automated model building [105]. The primary objective in this field is to automatically build a model for a first-order logic formula when its refutation cannot be found. According to Herbrand's theorem, a formula is satisfiable if and only if it has a Herbrand model, thus it is sufficient to build only Herbrand models, which consist of infinite term sets in general. Various finite representations of such models are thus considered for the automation of model building process [106–109]. In particular, these works provide efficient algorithms for working with models represented by tree automata and their extensions. A comprehensive overview of computational representations of Herbrand models, their properties, expressiveness, and the efficiency of their procedures is given in [110; 111]. Although representations proposed in these works can be used to represent invariants over ADTs, the design of algorithms for inferring such invariants remains a challenging task that has not been addressed in these studies.

The problem of finite term set representation is also stated in the context of formal tree languages as a task of designing extensions of tree automata with closure and decidability of basic language operations discussed in this chapter. Tree languages are systematically investigated in the context of formal languages [112], and, in particular, there are numerous works proposing the integration of various types of synchronization into tree automata [79–84]. However, there are several limitations with the representations proposed in this area. On the one hand, most investigated languages are those with efficient (low-degree polynomial) parsing algorithms, which consequently have low expressiveness due to the computational restrictions. On the other hand, the proposed classes of tree languages are usually not closed under cer-

tain Boolean operations, such as negation and intersection, which makes the task of adapting these classes for inductive invariant inference even more challenging.

Works focusing on extending tree automata with SMT constraints from other theories to so-called *symbolic tree automata* deserve separate mention [113; 114]. The class of invariants built on such automata could enable checking the satisfiability of Horn clause systems over a combination of ADTs with other SMT theories, as noted in [**10.1007/978-3-030-88806-0_20**]. The authors of this work initiated the adaptation of symbolic automata to the task of satisfiability checking for Horn clause systems no top of the ICE framework, implementing a teacher for this class of invariants. Further exploration of the class of invariants built on symbolic tree automata in the context of automatic invariant inference seems particularly promising.

Therefore, finite representations of tuple sets presented in works from these areas can serve as a foundation for future classes of inductive invariants over ADTs. Since many of them are constructed as extensions of the classes examined in this work, the methods of invariant inference proposed in this work can also be adapted to infer invariants in these new classes.

5.4 Conclusion

Among all classes of program invariants for which effective automatic invariant inference procedures exist, the most expressive ones are $\text{Reg}_\times$ and ElemReg. They allow both complex recursive relationships and synchronous relationships to be expressed, therefore, they extend the applicability of the automatic invariant inference in practice. However, due to the high expressive power, the automatic invariant inference in these classes can be difficult due to the growing complexity of primitive operations. The next chapter compares the effectiveness of existing and proposed methods of invariant inference for the considered classes.

Chapter 6. Implementation, Related Work and Evaluation

6.1 Pilot Implementation

All the approaches proposed in this work are implemented in a Horn solver RInGen (*R*egular *In*variant *Gen*erator)[1]. The implementation comprises 5200 lines of F# code. A Horn solver is developed from scratch as proposed algorithms require non-trivial manipulations of formulas and the results of other logical solvers.

The overall architecture of the tool is presented in Figure 6.1. RInGen takes as input a CHC system in the SMTLIB2 format [115]. The CHC system is parsed and simplified: equalities, selectors, and testers are eliminated. Then, depending on the options submitted to the solver, one of the algorithms proposed in this work is started. A "Substitution of ADT with uninterpreted functions module" implements the algorithm from the Chapter 2, and the "Tree automata first-order declaration generator" implements the algorithm from the Chapter 3. The output of each of these algorithms is a formula over uninterpreted functions. It is passed to an external logical solver — either Vampire automated theorem prover or cvc5 SMT-solver. As a result, RInGen returns a safe inductive invariant if the CHC system is satisfiable, otherwise, it returns a resolution refutation.

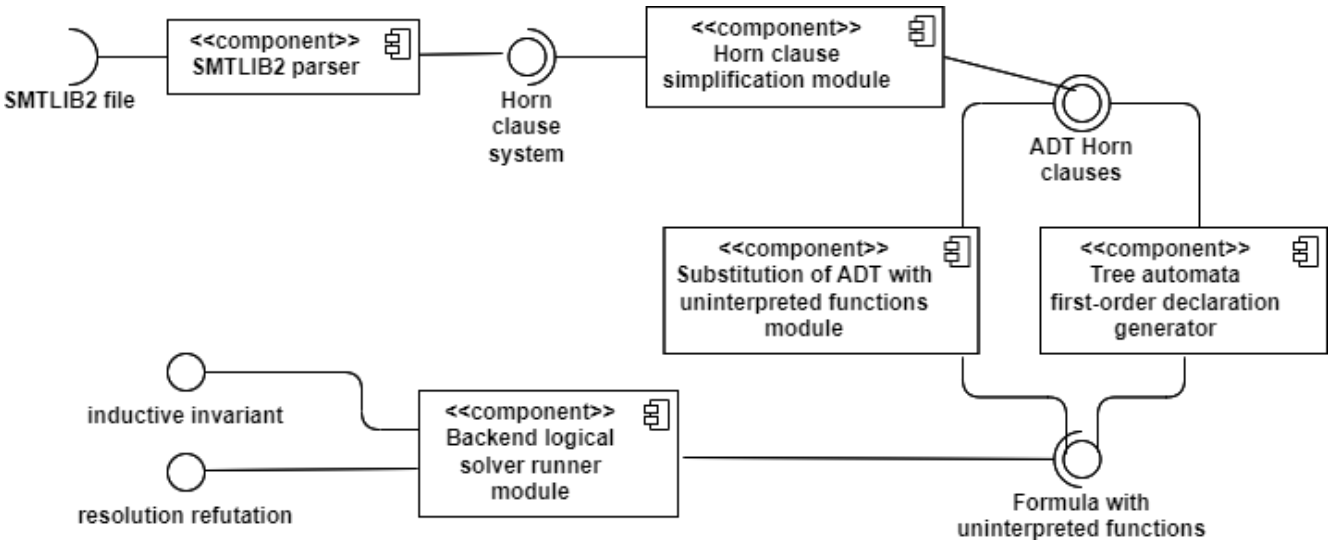


Figure 6.1 – Architecture of RInGen

RInGen. That is, we implemented the approach presented in Chapter 2 within the RInGen Horn solver. Both the approach itself and its implementation

[1] https://github.com/Columpio/RInGen

involve the use of an external solver $\mathcal{V}$ for the theory of uninterpreted functions with quantifiers, therefore the proposed implementation will be further denoted as RInGen($\mathcal{V}$). Specifically, Vampire [74] and the SMT solver cvc5 are used as the $\mathcal{V}$ in the experiments. Vampire uses a portfolio-based approach [116]: it iterates through various satisfiability checking techniques, mostly based on saturation of the system [94] and finite model finding [90]. cvc5 is used in the finite model finding mode[2] [89]. Both tools can prove the satisfiability of the system and finding counterexamples.

RInGen-Sync. The approach presented in Chapter 3 is implemented as an extension of RInGen($\mathcal{V}$). In the experiments, cvc5 is used as the external solver $\mathcal{V}$, as RInGen-Sync generates symbols with high arity, which are not supported by Vampire[3]. Thus, the implementation will be hereinafter referred to as RInGen-Sync[4].

RInGen-CICI. The approach presented in Chapter 4 is implemented within the codebase of the Horn solvers Racer [23] (a successor of the Spacer Horn solver [22], implemented in the logical solver Z3[5]) and the Horn solver RInGen($\mathcal{V}$)[6]. This implementation will be referred to as RInGen-CICI($\mathcal{V}$). Both parts of this implementation in the Horn solvers Racer and RInGen($\mathcal{V}$) are described below. These Horn solvers will be referred to as *basic* with respect to the RInGen-CICI($\mathcal{V}$) Horn solver.

The Racer Horn solver is developed by Arie Gurfinkel and Hari Govind Vediramana Krishnan from the University of Waterloo. It is based on an approach called Property-Directed Reachability (PDR) [22], which can be viewed as a complex instance of CEGAR. PDR builds abstract states in the form of conjunctions of formulas (called *lemmas*) at various *levels* by iteratively increasing the level in a loop. The following properties of lemma sets are maintained: if a set of lemmas $\{\varphi_i\}$ is constructed at level n, then $\bigwedge_i \varphi_i$ over-approximates all states reachable in less than n transition steps, and under-approximates the safety property. Thus, PDR lemmas fulfill the requirement of abstraction in the Collaborate procedure (see Listing 4.4). Racer is modified to asynchronously pass the set of lemmas from the last level to a new process of RInGen($\mathcal{V}$) at the end of each iteration.

[2] with the `--finite-model-find` option

[3] https://github.com/vprover/vampire/issues/348#issuecomment-1091782513

[4] https://github.com/Columpio/RInGen/releases/tag/ringen-tta

[5] https://github.com/Columpio/z3/tree/racer-solver-interaction

[6] https://github.com/Columpio/RInGen/releases/tag/chccomp22

The COLLABORATE procedure (see Listing 4.4) is implemented in the RINGEN($\mathcal{V}$). The following generalization of the RESIDUALCHCS($\mathcal{P}, a$) procedure (see 4.2.3) is implemented. The conjunctive form of lemmas from RACER is used to infer invariants with a more general shape: $\bigwedge_i(\varphi_i(\overline{x}) \vee \overline{x} \in L_i)$. Hence, given $a(P) = \bigwedge_i \varphi_i$, we replace all atoms $P(\overline{t})$ with a *conjunction of disjunctions* $\bigwedge_i(\varphi_i(\overline{t}) \vee L_i(\overline{t}))$ with new predicate symbols L_i. This allows inferring more general invariants than those from the union of ELEM and $\mathcal{A}$ (see Definition 25), which consists only of formulas of the form $\varphi(\overline{x}) \vee \overline{x} \in L$.

After the transformations, the RINGEN($\mathcal{V}$) calls the external solver $\mathcal{V}$ with a 30 second time limit. Then its results are passed back to RACER, where they are processed asynchronously. Moreover, the implementation does not perform the costly CNF transformation from Listing 4.5, as the RINGEN($\mathcal{V}$) takes Horn clauses in arbitrary form, since it relies on the external solver $\mathcal{V}$ with full first-order logic support.

6.2 Related Work

This section is dedicated to the comparison of proposed methods for solving Horn clause systems with algebraic data types and existing methods, implemented in tools such as SPACER, RACER, ELDARICA, VERICAT, HOICE, and RCHC. We selected only the tools supporting Horn clause systems over algebraic data types that verify both the satisfiability and unsatisfiability of these systems. For instance, tools addressing the related problem of automating induction for theorems with algebraic data types, such as, for example, CVC5 in induction mode [117], ADTIND [118] and others are not considered, as they do not accept Horn clause systems as input. Also, logic programming tools (such as PROLOG [119]) are not considered because they only check the unsatisfiability of Horn clause systems and cannot show their satisfiability.

Table 6.1 presents the comparison of Horn solvers: existing ones (upper block) and those proposed in this work (lower block). The proposed Horn solvers are described in previous Section 6.1, and the methods they implement are described in Chapters 2, 3, and 4 of this work. The word "Transf." indicates that the tool is built using non-trivial transformations of the system; "FMF" denotes the application of automatic finite-model finding (e. g., see [89; 90]); a dash in the "Invariant class" column means the following: although the output of the tool implicitly encodes its inductive invariant when the system is satisfiable, there is no always halting

Table 6.1 – Comparison of Horn solvers with ADT support

Tool	Invariant class	Method	Returns the invariant	Fully automatic
Spacer	Elem	IC3/PDR	Yes	Yes
Racer	CatElem	IC3/PDR	No	No
Eldarica	SizeElem	CEGAR	Yes	Yes
VeriCaT	–	Transf.	No	Yes
HoIce	Elem	ICE	Yes	Yes
RCHC	Reg_{+}	ICE	Yes	Yes
RInGen(cvc5)	Reg	Transf. + FMF	Yes	Yes
RInGen(Vampire)	–	Transf. + Saturation	No	Yes
RInGen-Sync	$\text{Reg}_{\times}$	Transf. + FMF	Yes	Yes
RInGen-CICI(cvc5)	ElemReg	CEGAR($\mathcal{O}$)	Yes	Yes
RInGen-CICI(Vampire)	–	CEGAR($\mathcal{O}$)	No	Yes

procedure that can check this output. The other designations are explained in the subsections dedicated to corresponding tools.

Inductive invariant classes of most of the examined tools differ. A comparison of these invariants classes is given in Chapter 5. It is important for comparing tools because if a tool infers invariants in a certain class, then the expressiveness problem of this class (the inability to express certain types of relations) becomes the problem of non-termination of this tool. In other words, since none of the existing tools checks whether for a given Horn clause system there is an invariant in its class *at all*[7], then the tool will not terminate in case of the absence of the invariant.

Further on, we provide a comparative description of the existing tools.

The Spacer tool [22] constructs elementary models (the Elem class). This tool is based on a classic satisfiability procedure for ADTs, as well as interpolation and quantifier elimination procedures [122]. At its core, the tool employs a technique called *property-directed reachability* (IC3/PDR), which evenly distributes analysis time between counterexample search and safe inductive invariant inference, propagating information about reachability of unsafe properties and partial safety lemmas.

[7] On the one hand, this task is as complex as the verification task itself; on the other hand, so far only a few studies have been dedicated to it (see, for example, [120; 121])

The tool can infer invariants in a combination of algebraic and other data types, and returns verifiable certificates. The approach used in the tool is both sound and complete. A drawback of the tool is that it expresses invariants in the constraint language, and therefore often does not terminate on problems with ADTs.

The RACER tool [23] is an evolution of the SPACER tool. It can infer invariants in the constraint language extended with catamorphisms. This constraint language is denoted in the Table 6.1 as CATELEM. RACER inherits all the advantages of the SPACER approach. A drawback of the approach is that it is not fully automatic, as it requires manually specifying catamorphisms, which can be challenging in practice because it can be hard to guess which catamorphisms will be required for the invariant of the given problem. A drawback of the tool is that it does not return any verifiable certificates.

The ELDARICA tool [24] constructs models with term size constraints, which calculate the total number of constructor occurrences (the SIZEELEM class). This extension is very limited, as the introduced function counts all constructors at once, thus it cannot express many properties, such as properties depending on the tree height. The ELDARICA tool employs the CEGAR with predicate abstraction and an embedded SMT solver PRINCESS [73], which provides a satisfiability and an interpolation procedures for algebraic data types with term size constraints. These procedures are based on the reduction of this theory to a combination of theories of uninterpreted functions and linear arithmetic [100].

The VERICAT tool [29–32] takes a CHC system over theories of linear arithmetic and ADTs and completely eliminates ADTs from the original system of Horn clauses by folding, unfolding, introducing new clauses, and other syntactic transformations. It produces a Horn clause system without ADTs, on which any efficient Horn solver, such as SPACER or ELDARICA, can be run. The main advantage of this approach is that it is designed to work with problems where algebraic data types are combined with other theories. The main drawbacks of the approach are as follows: the transformation process itself may not terminate, and due to the transformation, it is impossible to recover the invariant of the original system, i. e., the tool does not return a verifiable certificate.

The HoIce tool [25] constructs elementary invariants using a machine learning approach called ICE [52]. Its advantages include the ability to infer invariants for combinations of ADTs with other theories, as well as correctness and soundness, and finally, the ability to return verifiable correctness certificates. Its disadvantage is that it produces invariants in an inexpressive constraint language, and thus often does not terminate.

The RCHC tool [26; 123] also uses the ICE approach; it expresses inductive invariants of programs over ADTs using *tree automata* [33]. However, due to the complexities of expressing tuples of terms with automata, described in Section 5.2.1, the approach is often inapplicable even for the simplest examples where classical symbolic invariants exist.

Conclusions. Compared to the method of RCHC, the approaches proposed in this thesis provide alternative ways of inferring regular invariants and their superclasses. Therefore, they can be combined with the RCHC approach to converge faster, if the inductive invariant exists. Compared to the methods of the remaining tools, the proposed approaches can infer invariants in independent classes of regular invariants. Therefore, the application of the proposed methods in conjunction with existing ones can solve a wider set of problems.

6.3 Evaluation

6.3.1 Tool Selection

We have selected the Racer [23] and Eldarica [24] tools for compassion, as they are Horn solvers with ADT support leading in the CHC-COMP competition [28]. We also selected cvc5-Ind (cvc5 in induction mode) [117] and VeriCaT [29]. Although these tools do not build inductive invariants explicitly, which makes it impossible to check their correctness, their runs on an equivalent benchmark is added to the experimental comparison as they solve a related problem.

A HoIce Horn solver [25] is not included in the evaluation since it is not faster than Racer [23] and it infers invariants in the same class Elem. A RCHC Horn solver [26] is not included in the evaluation because it is unstable and often fails and returns incorrect results.

6.3.2 Benchmark Suite

The experiments are conducted on the TIP (Tons of Inductive Problems) benchmark [124], which is the test set from the CHC-COMP 2022 competition ADT track[8]. The TIP benchmark consists of 454 CHC systems derived from Haskell programs with ADTs and recursion. The benchmark includes such algebraic data types as lists, queues, regular expressions, and Peano integers.

6.3.3 Setup

The experiments are conducted on the StarExec platform [125]: a cluster of machines with Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz and Red Hat Enterprise Linux 7[9]. A CPU runtime limit for each tool is 600 seconds and a memory limit is 16 GB.

6.3.4 Research Questions

We have posed the following research questions for the experiments.

Research question 1 (Number of solutions). Since the main goal of this work is to propose approaches for checking the satisfiability of more systems than analogues by inductive invariant inference, the following questions are the most important.

- Do the proposed methods show satisfiability of more systems than approaches that infer classical symbolic invariants?
- Do the proposed methods also show satisfiability of systems that have classical invariants?

Research question 2 (Performance).

- What is the performance of RInGen and RInGen-Sync on problems that they, as well as existing tools, are able to solve?
- Collaborative inference in RInGen-CICI may require parallel running of multiple oracle instances. What is the impact of parallel running on performance?

Research question 3 (Significance of the inductive invariant class). Collaborative inference in RInGen-CICI can theoretically accelerate the convergence of

[8] https://github.com/chc-comp/ringen-adt-benchmarks
[9] https://www.starexec.org/starexec/public/machine-specs.txt

Table 6.2 – Evaluation results. SAT indicates that the system is satisfiable (there is an inductive invariant), UNSAT indicates that the system is unsatisfiable.

Tool	SAT	UNSAT
Racer	26	22
Eldarica	46	12
VeriCaT	16	10
cvc5-Ind	0	13
RInGen(cvc5)	25	21
RInGen(Vampire)	135	46
RInGen-Sync	43	21
RInGen-CICI(cvc5)	117	19
RInGen-CICI(Vampire)	189	28

the search for classical symbolic invariants. What is the share of classical symbolic invariants in all problems, uniquely solved by RInGen-CICI?

6.4 Results

6.4.1 Number of Solutions

The number of problems from the benchmark suite solved by the existing and proposed tools is presented in Table 6.2. Existing tools are placed above the line, while proposed tools are placed below it.

RInGen. On all 12 problems where Eldarica returned UNSAT, RInGen terminated with the same result, and RInGen found more counterexamples. The RInGen(cvc5), RInGen(Vampire), and Racer found counterexamples for 21, 46, and 22 clause systems, respectively, with each of them finding several unique counterexamples. RInGen(Vampire) found significantly more UNSAT results than other tools, as Vampire implements an efficient refutation inference procedure. Therefore, even though the proposed algorithms are designed to infer more inductive invariants, they also can find unique counterexamples. Next, Eldarica found 46 invariants in contrast to 25 and 135 invariants found by RInGen(cvc5) and RInGen(Vampire). Out of these, Eldarica solves 25 unique (not solved by RInGen(cvc5)) benchmarks, each of which is a specification of some property of order predicates (i. e., $<$, $\leqslant$, $>$, $\geqslant$) on Peano integers. These problems are easily solved by Eldarica, as order predicates are included in the SizeElem invariant

class as primitives. However, RInGen(cvc5) solves 13 unique (not solved by Eldarica) problems, whose invariants are not expressible in the invariant class of the Eldarica. The effectiveness of the approach implemented in the RInGen heavily depends on the external solver, as evidenced by the fact that RInGen(Vampire) inferred 5 times more invariants than RInGen(cvc5).

RInGen-Sync. This tool terminated with UNSAT on 21 problems, these results exactly match the 21 results of the RInGen(cvc5), since RInGen-Sync inherits the counterexample search from the latter.

Among all SAT results obtained by Eldarica, 38 are also obtained by RInGen-Sync. The large overlap with the results of Eldarica is due to the fact that Eldarica deals well with problems encoding the order on Peano numbers, which are also well-encoded by the synchronous tree automata with full convolution used in RInGen-Sync. Racer halted with a SAT result on 26 systems, 15 of which intersect with the results of RInGen-Sync. Additionally, RInGen-Sync inferred 4 unique invariants. Despite the theoretical expressive power of the synchronous tree automata with full convolution used in RInGen-Sync, which should yield a larger number of solutions, a finite model finder that RInGen-Sync uses as a backend does not terminate on problems with a large number of quantifiers, and therefore RInGen-Sync often does not terminate. The results do not change if we increase the time limit to 1200 seconds or switch the backend from cvc5 to other finite model finding tools, like Vampire in appropriate mode. The small overlap with Racer suggests that although the class of elementary invariants is theoretically almost fully contained in the class of synchronous regular invariants, in practice the proposed approach does not effectively infer invariants of systems that have elementary invariants.

RInGen-CICI. RInGen-CICI solves fewer *unsafe problems* than the best of the basic solvers: RInGen-CICI obtained 19 (with cvc5) and 28 (with Vampire) UNSAT results against 21 (with cvc5) and 46 (with Vampire) UNSAT results obtained by RInGen. The main reason is that the proposed approach is designed to solve the more complex task of inferring inductive invariants and does not change the operation of the basic counterexample finding algorithms. That is, our approach can be integrated with orthogonal improvements to counterexample search, for example, those proposed in [126]. Thus, all counterexamples obtained by RInGen-CICI are directly obtained from one of the basic solvers. Some counterex-

amples found by RInGen are not found by RInGen-CICI, as it runs RInGen with a time limit of 30 seconds.

It is important to note that all 20 SAT and 15 UNSAT answers obtained by Racer, are also obtained by RInGen-CICI, with the exception of one UNSAT answer.

On safe problems, RInGen-CICI outperformed basic solvers: RInGen-CICI(cvc5) obtained 117 SAT responses, while Racer obtained 20 SAT responses, and RInGen(cvc5) 25. RInGen-CICI(Vampire) obtained 189 SAT responses, with 20 SAT responses from Racer and 135 from RInGen(Vampire). Thus, RInGen-CICI solves significantly more SAT tasks than the basic tools working separately: 117 versus $20 + 25$ and 189 versus $20 + 135$ for the respective cvc5 and Vampire backends. In particular, RInGen-CICI(cvc5) solves 97 tasks not solved by Racer and 94 tasks not solved by RInGen(cvc5). RInGen-CICI(Vampire) solves 169 tasks not solved by Racer and 60 tasks not solved by RInGen(Vampire). Therefore, the collaborative invariant inference method shows the satisfiability of significantly more systems than the parallel launch of basic tools.

However, there are problems that are solved by the basic solvers but not by the proposed tool. RInGen-CICI(cvc5) does not solve 7 problems that are successfully solved by RInGen(cvc5). Two of these problems could be solved by the proposed tool if the 30-second time limit in RInGen-CICI is increased. Existing verification time prediction methods, such as [127], can be applied to avoid hard-coding a time limit. The remaining 5 problems are solved instantly, but their results cannot be retrieved from the inter-process interaction in the implemented solution. The reason is that Racer spends too much time solving SMT constraints and therefore does not read the results of the backend solver. This technical issue can be avoided by reading the results of the backend solver at more frequent checkpoints, which, however, will result in overhead increase. The same goes for RInGen-CICI(Vampire), which failed to solve 24 problems solved by the basic solvers. Only 8 of them are unsolved due to the low time limit for the backend, while the remaining 16 are due to Racer divergence in solving SMT constraints.

Finally, RInGen(cvc5) does not outperform the existing solutions, but it provides many unique solutions compared to them, as it inferred invariants in a new class. RInGen(Vampire), built on the same approach, solves over 2.5 times more problems than the best of the existing tools, for the same reason. Despite the

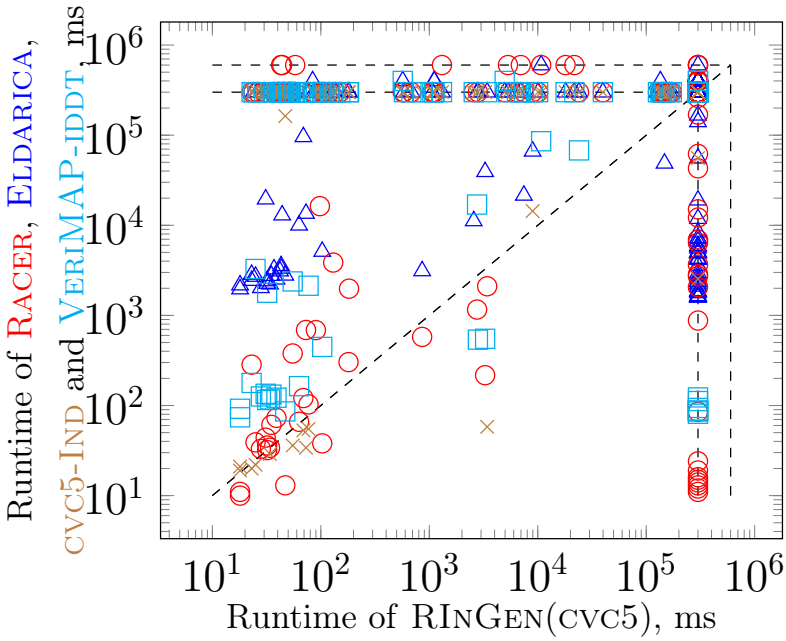


Figure 6.2 – Performance comparison. Each point on the plot represents a pair of runtimes.

fact that the class of invariants of RInGen-Sync is significantly wider, inferring invariants in it is much harder. Although it infers almost twice as many invariants as RInGen(cvc5), it does not outperform the best of the existing tools. The best results are shown by RInGen-CICI(cvc5), which solves 235% more tasks than the parallel composition of Racer and RInGen(cvc5), and also 39% more tasks with the Vampire backend, thanks to the balance between the expressiveness of the invariant class and the efficiency of the invariant inference procedure. The best of the proposed tools, RInGen-CICI(Vampire), solves a total of $189 + 28$ problems, which is about 3.74 times more than the best of the existing tools, Eldarica, which solves $46 + 12$ problems.

6.4.2 Performance

The plots in Figure 6.2 show that RInGen(cvc5) not only infers more invariants but also works faster than the other tools by one order of magnitude on average. In the figure, some unsafe systems are solved faster by cvc5-Ind, VeriCaT, and Racer. The reasons for that could be a more efficient quantifier instantiation procedure in cvc5-Ind and a more balanced trade-off between invariant inference and counterexample search in the core of Racer (which is also called by VeriCaT).

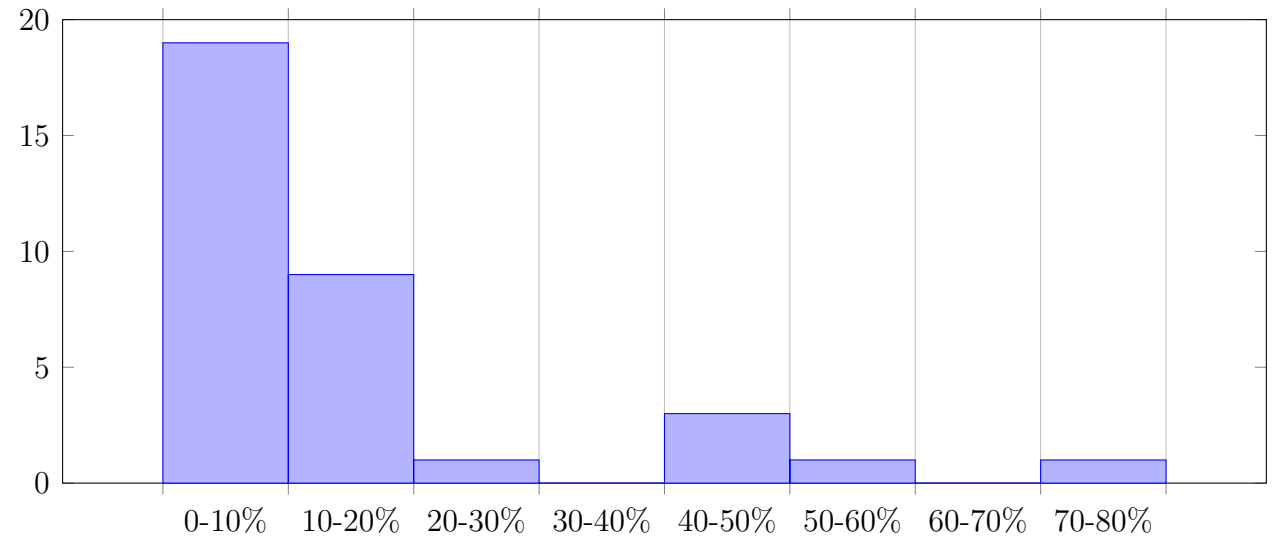


Figure 6.3 – Number of benchmark instances solved by both RInGen-CICI and Racer (y-axis), and the CPU time overhead (x-axis) of running RInGen-CICI compared to Racer. Racer outperforms RInGen-CICI on 34 instances. There are no instances with an overhead larger than 80%, so the x-axis is not shown further.

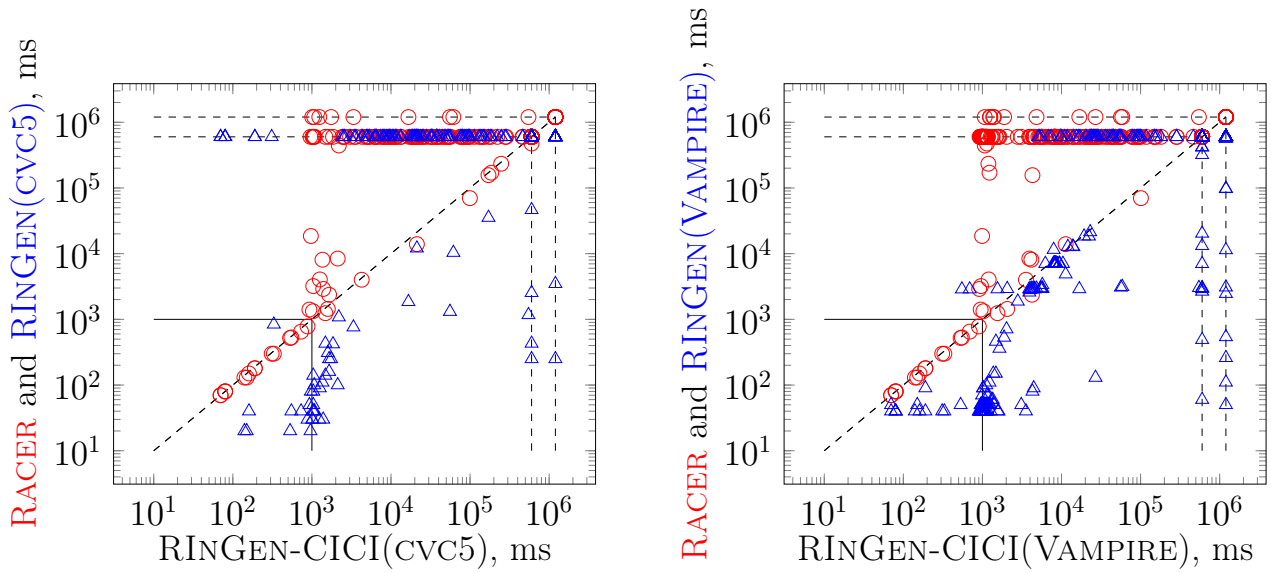


Figure 6.4 – Runtime comparison

On problems that are solved by several tools, the RInGen(cvc5) operates two orders of magnitude faster on average. The RInGen(Vampire) runs took even less time than runs of the RInGen(cvc5).

RInGen-Sync and RInGen-CICI. Figure 6.4 shows the *overall performance plots* of RInGen-CICI compared with the basic solvers. Each point on the plot represents the running time (in milliseconds) of RInGen-CICI (x-axis) and the competing tool (y-axis): triangles represent RInGen, while circles represent

Racer. The outer dashed lines represent when a tool halts with an error. There are such runs for both Racer and RInGen-CICI, due to the instability of the used version of Racer[10]. The inner dashed lines denote cases when the solver reached the time limit. Since RInGen-CICI solves significantly more instances than the competing solvers, most of the figures are on the upper dashed lines of both graphs. Half of the remaining figures are near the diagonal, meaning that the collaborative operation finished after the first collaborative solver call. The other half of the figures are near one second (which is marked by a solid line in the bottom left corner) for the same reason why some problems are not solved: the internal engine of Racer in RInGen-CICI is solving complex SMT constraints and therefore does not read the result of the backend for some time. Most of the circles that do not hit the dashed lines are near the diagonal on both plots, meaning that RInGen-CICI worked comparably with Racer on problems that are solved by both tools.

The chart in Figure 6.3 shows the *overhead* of collaborative inference in RInGen-CICI. There are only 34 and 35 problems solved simultaneously by Racer and RInGen-CICI(cvc5) and Racer and RInGen-CICI(Vampire), respectively. In 35 out of these 69 runs, RInGen-CICI is faster than Racer, but in the remaining 34, no backend call is successful, so RInGen-CICI behaved just like Racer, but with the overhead of process creation. The overhead for these 34 runs is shown in Figure 6.3. The chart shows how many times slower is RInGen-CICI compared to Racer. Overhead in most runs is close to 10%: the average overhead across all runs is 15%, and the median is 8%. Overhead exceeded 20% in only 6 runs. In three of them, Racer operates from 14 to 70 seconds, and RInGen-CICI is 40-50% slower due to the accumulated number of concurrently running interactive processes. The other runs with overhead more than 20% are those where Racer operates no more than 2 seconds, and RInGen-CICI from 2 to 4 seconds. This results in a high percentage, which thus can be discarded.

Concluding the answer to research question 2, note that the median overhead of RInGen-CICI is about 8%. High overhead (>50%) is observed only in six runs.

6.4.3 Significance of the Inductive Invariant Class

It is hard to *precisely* count which of the problems solved only by RInGen-CICI do not belong to the Elem invariant class, as the task of formally

[10]We used a particular version in the experiments because it gives the same number of solved problems on the benchmarks compared to the stable version, but sometimes works almost ten times faster.

proving inexpressibility in Elem is hard even for a human. However, the number of such problems can be estimated as the number of those problems where the invoked solver returns either a tree automaton with loops or saturation; all unique problems with a SAT result obtained by RInGen-CICI fit this criterion. This implies that all invariants of problems uniquely solved by RInGen-CICI do not belong to the Elem invariant class. Thus, the main reason for the success of the RInGen-CICI tool compared to other tools is the expressiveness of the class of inductive invariants it uses.

Conclusion

The core results of the thesis are as follows.

1. We have proposed an efficient method for automatic inductive invariant inference based on tree automata. With that, these invariants can express recursive relationships across a broad spectrum of real-world programs. The method relies on finite model search.
2. We have proposed a method for automatic inductive invariant inference based on program transformation and finite model search within the invariant class based on synchronous tree automata. This class of invariants allows expressing recursive and synchronous relations.
3. We have proposed a class of inductive invariants based on a Boolean combination of classical invariants and tree automata, which, on the one hand, allows to express recursive relations in real programs, and, on the other hand, allows to effectively infer inductive invariants. We have also proposed an efficient method of combined inductive invariant inference in this class, which infers invariants in the combined subclasses.
4. We have conducted a theoretical comparison of existing and proposed classes of inductive invariants, including the formulation and proof of pumping lemmas for the constraint language and for the constraint language extended with the term size function, which allow to prove the inexpressibility of an invariant in the constraint language.
5. We have completed a pilot software implementation of the proposed methods in the F# language as part of the RInGen tool; we have then compared this tool with existing methods on a commonly accepted test set of functional program verification tasks "Tons of Inductive Problems": the implementation of the best of the proposed methods was able to solve 3.74 times more tasks in the allotted time than the best of the existing tools.

Concerning the **recommendations for applying the thesis results** in industry and scientific research, the developed methods are applicable for automating reasoning about Horn clause systems over the theory of algebraic data types, and that their implementation is made in a publicly available tool RInGen. The created tool can be used as a main component for verification in static code analyzers and

verifiers for languages with algebraic data types, such as Rust, Scala, Solidity, Haskell and OCaml. The tool can also be used to prove unreachability of errors or specified code fragments, which are important tasks for computer security and quality assurance.

FInally, we have also defined the **prospects for further development of the topic**, the main one of which is the extension of the proposed classes of inductive invariants and methods of their inference to combinations of algebraic data types with other data types common in programming languages, such as integers, arrays, and strings. This will allow to infer invariants of programs with complex functional relationships between structures and the data contained therein, which will significantly expand the practical applicability of the proposed methods.

References

1. Symbolic model checking [Text] / E. Clarke [et al.] // Computer Aided Verification / Ed. by R. Alur, T. A. Henzinger. — Berlin, Heidelberg: Springer Berlin Heidelberg, 1996. — P. 419–422.
2. *Godefroid, P.* SAGE: Whitebox Fuzzing for Security Testing: SAGE Has Had a Remarkable Impact at Microsoft. [Text] / P. Godefroid, M. Y. Levin, D. Molnar // Queue. — New York, NY, USA, 2012. — Vol. 10, № 1. — P. 20–27. — URL: https://doi.org/10.1145/2090147.2094081.
3. *Wohrer, M.* Smart contracts: security patterns in the ethereum ecosystem and solidity [Text] / M. Wohrer, U. Zdun // 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE). — 2018. — P. 2–8.
4. *Floyd, R. W.* Assigning meanings to programms [Text] / R. W. Floyd // Proccedings of the AMS Symposium on Appllied Mathematics. Vol. 19. — American Mathematical Society, 1967. — P. 19–31.
5. *Hoare, C. A. R.* An Axiomatic Basis for Computer Programming [Text] / C. A. R. Hoare // Commun. ACM. — New York, NY, USA, 1969. — Vol. 12, № 10. — P. 576–580. — URL: https://doi.org/10.1145/363235.363259.
6. *Rushby, J.* Subtypes for specifications: predicate subtyping in PVS [Text] / J. Rushby, S. Owre, N. Shankar // IEEE Transactions on Software Engineering. — 1998. — Vol. 24, № 9. — P. 709–720.
7. Flux: Liquid Types for Rust [Text] / N. Lehmann [et al.]. — 2022. — URL: https://arxiv.org/abs/2207.04034.
8. *Suter, P.* Satisfiability Modulo Recursive Programs [Text] / P. Suter, A. S. Köksal, V. Kuncak // Static Analysis / Ed. by E. Yahav. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. — P. 298–315.
9. *Leino, K. R. M.* Dafny: An Automatic Program Verifier for Functional Correctness [Text] / K. R. M. Leino // Logic for Programming, Artificial Intelligence, and Reasoning / Ed. by E. M. Clarke, A. Voronkov. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. — P. 348–370.

10. *Filliâtre, J.-C.* Why3 — Where Programs Meet Provers [Text] / J.-C. Filliâtre, A. Paskevich // Programming Languages and Systems / Ed. by M. Felleisen, P. Gardner. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. — P. 125–128.

11. *Müller, P.* Viper: A Verification Infrastructure for Permission-Based Reasoning [Text] / P. Müller, M. Schwerhoff, A. J. Summers // Verification, Model Checking, and Abstract Interpretation / Ed. by B. Jobstmann, K. R. M. Leino. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2016. — P. 41–62.

12. Dependent Types and Multi-Monadic Effects in F* [Text] / N. Swamy [et al.] // SIGPLAN Not. — New York, NY, USA, 2016. — Vol. 51, № 1. — P. 256–270. — URL: https://doi.org/10.1145/2914770.2837655.

13. The Coq Proof Assistant : Reference Manual : Version 7.2 [Text]: tech. rep. / B. Barras [et al.]; INRIA. — 2002. — P. 290. — RT–0255. — URL: https://inria.hal.science/inria-00069919.

14. *Brady, E.* Idris, a general-purpose dependently typed programming language: Design and implementation [Text] / E. Brady // Journal of Functional Programming. — 2013. — Vol. 23, № 5. — P. 552–593.

15. *Vezzosi, A.* Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types [Text] / A. Vezzosi, A. Mörtberg, A. Abel // Proc. ACM Program. Lang. — New York, NY, USA, 2019. — Vol. 3, ICFP. — URL: https://doi.org/10.1145/3341691.

16. *Moura, L. d.* The Lean 4 Theorem Prover and Programming Language [Text] / L. d. Moura, S. Ullrich // Automated Deduction – CADE 28 / Ed. by A. Platzer, G. Sutcliffe. — Cham: Springer International Publishing, 2021. — P. 625–635.

17. *Makowsky, J.* Why horn formulas matter in computer science: Initial structures and generic examples [Text] / J. Makowsky // Journal of Computer and System Sciences. — 1987. — Vol. 34, № 2. — P. 266–292. — URL: https://www.sciencedirect.com/science/article/pii/0022000087900274.

18. Synthesizing Software Verifiers from Proof Rules [Text] / S. Grebenshchikov [et al.] // Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. — Beijing, China: Association for Computing Machinery, 2012. — P. 405–416. — (PLDI '12). — URL: https://doi.org/10.1145/2254064.2254112.

19. Horn Clause Solvers for Program Verification [Text] / N. Bjørner [et al.] // Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday / Ed. by L. D. Beklemishev [et al.]. — Cham: Springer International Publishing, 2015. — P. 24–51. — URL: https://doi.org/10.1007/978-3-319-23534-9_2.

20. *Matsushita, Y.* RustHorn: CHC-Based Verification for Rust Programs [Text] / Y. Matsushita, T. Tsukada, N. Kobayashi // ACM Trans. Program. Lang. Syst. — New York, NY, USA, 2021. — Vol. 43, № 4. — URL: https://doi.org/10.1145/3462205.

21. SolCMC: Solidity Compiler's Model Checker [Text] / L. Alt [et al.] // Computer Aided Verification / Ed. by S. Shoham, Y. Vizel. — Cham: Springer International Publishing, 2022. — P. 325–338.

22. *Komuravelli, A.* SMT-based model checking for recursive programs [Text] / A. Komuravelli, A. Gurfinkel, S. Chaki // Formal Methods in System Design. — 2016. — Vol. 48, № 3. — P. 175–205.

23. *K, H. G. V.* Solving Constrained Horn Clauses modulo Algebraic Data Types and Recursive Functions [Text] / H. G. V. K, S. Shoham, A. Gurfinkel // Proc. ACM Program. Lang. — New York, NY, USA, 2022. — Vol. 6, POPL. — URL: https://doi.org/10.1145/3498722.

24. *Hojjat, H.* The ELDARICA Horn Solver [Text] / H. Hojjat, P. Rümmer // 2018 Formal Methods in Computer Aided Design (FMCAD). — 2018. — P. 1–7.

25. *Champion, A.* HoIce: An ICE-Based Non-linear Horn Clause Solver [Text] / A. Champion, N. Kobayashi, R. Sato // Programming Languages and Systems / Ed. by S. Ryu. — Cham: Springer International Publishing, 2018. — P. 146–156.

26. *Haudebourg, T.* Automatic verification of higher-order functional programs using regular tree languages [Text]: PhD thesis / Haudebourg Timothée. — 2020. — URL: http : / / www . theses . fr / 2020REN1S060 / document; 2020REN1S060.

27. Verifying Catamorphism-Based Contracts using Constrained Horn Clauses [Text] / E. de Angelis [et al.] // Theory and Practice of Logic Programming. — 2022. — Vol. 22, № 4. — P. 555–572.

28. *Angelis, E. D.* CHC-COMP 2022: Competition Report [Text] / E. D. Angelis, H. G. V. K // Electronic Proceedings in Theoretical Computer Science. — 2022. — Vol. 373. — P. 44–62. — URL: https://doi.org/10.4204%2Feptcs.373.5.

29. Satisfiability of constrained Horn clauses on algebraic data types: A transformation-based approach [Text] / E. De Angelis [et al.] // Journal of Logic and Computation. — 2022. — Vol. 32, № 2. — P. 402–442. — eprint: https://academic.oup.com/logcom/article-pdf/32/2/402/42618008/exab090.pdf. — URL: https://doi.org/10.1093/logcom/exab090.

30. Analysis and Transformation of Constrained Horn Clauses for Program Verification [Text] / E. De Angelis [et al.] // Theory and Practice of Logic Programming. — 2022. — Vol. 22, № 6. — P. 974–1042.

31. Removing Algebraic Data Types from Constrained Horn Clauses Using Difference Predicates [Text] / E. De Angelis [et al.] // Automated Reasoning / Ed. by N. Peltier, V. Sofronie-Stokkermans. — Cham: Springer International Publishing, 2020. — P. 83–102.

32. Solving Horn Clauses on Inductive Data Types Without Induction [Text] / E. De Angelis [et al.] // Theory and Practice of Logic Programming. — 2018. — Vol. 18, № 3/4. — P. 452–469.

33. Tree Automata Techniques and Applications [Text] / H. Comon [et al.]. — 2008. — P. 262. — URL: https://hal.inria.fr/hal-03367725.

34. Автоматическое доказательство корректности программ с динамической памятью [Text] / Ю. О. Костюков [и др.] // Труды Института системного программирования РАН. — 2019. — Т. 31, № 5. — С. 37–62.

35. *Kostyukov, Y.* Beyond the Elementary Representations of Program Invariants over Algebraic Data Types [Text] / Y. Kostyukov, D. Mordvinov, G. Fedyukovich // Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. — Virtual, Canada: Association for Computing Machinery, 2021. — P. 451–465. — (PLDI 2021). — URL: https://doi.org/10.1145/3453483.3454055.

36. *Kostyukov, Y.* Collaborative Inference of Combined Invariants [Text] / Y. Kostyukov, D. Mordvinov, G. Fedyukovich // Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning. Vol. 94 / Ed. by R. Piskac, A. Voronkov. — EasyChair, 2023. — P. 288–305. — (EPiC Series in Computing). — URL: https://easychair.org/publications/paper/GRNG.

37. Генерация слабейших предусловий программ с динамической памятью в символьном исполнении [Text] / А. В. Мисонижник [и др.] // Научно-технический вестник информационных технологий, механики и оптики. — 2022. — Т. 22, № 5. — С. 982–991.

38. On computable numbers, with an application to the Entscheidungsproblem [Text] / A. M. Turing [et al.] // J. of Math. — 1936. — Vol. 58, № 345–363. — P. 5.

39. *Rice, H. G.* Classes of Recursively Enumerable Sets and Their Decision Problems [Text] / H. G. Rice // Transactions of the American Mathematical Society. — 1953. — Vol. 74, № 2. — P. 358–366. — URL: http://www.jstor.org/stable/1990888 (visited on 12/03/2022).

40. *Clarke, E. M.* Design and synthesis of synchronization skeletons using branching time temporal logic [Text] / E. M. Clarke, E. A. Emerson // Logics of Programs / Ed. by D. Kozen. — Berlin, Heidelberg: Springer Berlin Heidelberg, 1982. — P. 52–71.

41. *Clarke, E. M.* The Birth of Model Checking [Text] / E. M. Clarke // 25 Years of Model Checking: History, Achievements, Perspectives. — Berlin, Heidelberg: Springer-Verlag, 2008. — P. 1–26. — URL: https://doi.org/10.1007/978-3-540-69850-0_1.

42. *Kautz, H.* Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search [Text] / H. Kautz, B. Selman // Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2. — Portland, Oregon: AAAI Press, 1996. — P. 1194–1201. — (AAAI'96).

43. Chaff: Engineering an Efficient SAT Solver [Text] / M. W. Moskewicz [et al.] // Proceedings of the 38th Annual Design Automation Conference. — Las Vegas, Nevada, USA: Association for Computing Machinery, 2001. — P. 530–535. — (DAC '01). — URL: https://doi.org/10.1145/378239.379017.

44. *Silva, J. P. M.* GRASP-a new search algorithm for satisfiability. [Text] / J. P. M. Silva, K. A. Sakallah // ICCAD. Vol. 96. — Citeseer. 1996. — P. 220–227.

45. *Tinelli, C.* A DPLL-Based Calculus for Ground Satisfiability Modulo Theories [Text] / C. Tinelli // Logics in Artificial Intelligence / Ed. by S. Flesca [et al.]. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. — P. 308–319.

46. *Stump, A.* CVC: A Cooperating Validity Checker [Text] / A. Stump, C. W. Barrett, D. L. Dill // Computer Aided Verification / Ed. by E. Brinksma, K. G. Larsen. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. — P. 500–504.

47. Symbolic Model Checking without BDDs [Text] / A. Biere [et al.] // Tools and Algorithms for the Construction and Analysis of Systems / Ed. by W. R. Cleaveland. — Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. — P. 193–207.

48. *Kurshan, R. P.* The Automata-Theoretic Approach [Text] / R. P. Kurshan. — Princeton: Princeton University Press, 1995. — URL: https://doi.org/10.1515/9781400864041.

49. Counterexample-Guided Abstraction Refinement [Text] / E. Clarke [et al.] // Computer Aided Verification / Ed. by E. A. Emerson, A. P. Sistla. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2000. — P. 154–169.

50. *McMillan, K. L.* Interpolation and SAT-Based Model Checking [Text] / K. L. McMillan // Computer Aided Verification / Ed. by W. A. Hunt, F. Somenzi. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. — P. 1–13.

51. *McMillan, K. L.* Applications of Craig Interpolants in Model Checking [Text] / K. L. McMillan // Tools and Algorithms for the Construction and Analysis of Systems / Ed. by N. Halbwachs, L. D. Zuck. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. — P. 1–12.

52. ICE: A Robust Framework for Learning Invariants [Text] / P. Garg [et al.] // Computer Aided Verification / Ed. by A. Biere, R. Bloem. — Cham: Springer International Publishing, 2014. — P. 69–87.

53. *Bradley, A. R.* SAT-Based Model Checking without Unrolling [Text] / A. R. Bradley // Verification, Model Checking, and Abstract Interpretation / Ed. by R. Jhala, D. Schmidt. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. — P. 70–87.

54. IC3 Modulo Theories via Implicit Predicate Abstraction [Text] / A. Cimatti [et al.] // Tools and Algorithms for the Construction and Analysis of Systems / Ed. by E. Ábrahám, K. Havelund. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. — P. 46–61.

55. *Hoder, K.* Generalized Property Directed Reachability [Text] / K. Hoder, N. Bjørner // Theory and Applications of Satisfiability Testing – SAT 2012 / Ed. by A. Cimatti, R. Sebastiani. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. — P. 157–171.

56. *Cook, S. A.* Soundness and Completeness of an Axiom System for Program Verification [Text] / S. A. Cook // SIAM Journal on Computing. — 1978. — Vol. 7, № 1. — P. 70–90. — eprint: https://doi.org/10.1137/0207005. — URL: https://doi.org/10.1137/0207005.

57. *Blass, A.* Inadequacy of Computable Loop Invariants [Text] / A. Blass, Y. Gurevich // ACM Trans. Comput. Logic. — New York, NY, USA, 2001. — Vol. 2, № 1. — P. 1–11. — URL: https://doi.org/10.1145/371282.371285.

58. *Blass, A.* Existential fixed-point logic [Text] / A. Blass, Y. Gurevich // Computation Theory and Logic / Ed. by E. Börger. — Berlin, Heidelberg: Springer Berlin Heidelberg, 1987. — P. 20–36. — URL: https://doi.org/10.1007/3-540-18170-9_151.

59. *Blass, A.* The Underlying Logic of Hoare Logic [Text] / A. Blass, Y. Gurevich // Bulletin of the European Association for Theoretical Computer Science. Vol. 70. — 2000. — P. 82–110. — URL: https://www.microsoft.com/en-us/research/publication/142-underlying-logic-hoare-logic/.

60. Proving correctness of imperative programs by linearizing constrained Horn clauses [Text] / E. De Angelis [et al.] // Theory and Practice of Logic Programming. — 2015. — Vol. 15, № 4/5. — P. 635–650.

61. Relational Verification Through Horn Clause Transformation [Text] / E. De Angelis [et al.] // Static Analysis / Ed. by X. Rival. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2016. — P. 147–169.

62. *Mordvinov, D.* Synchronizing Constrained Horn Clauses [Text] / D. Mordvinov, G. Fedyukovich // LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning. Vol. 46 / Ed. by T. Eiter, D. Sands. — EasyChair, 2017. — P. 338–355. — (EPiC Series in Computing). — URL: https://easychair.org/publications/paper/LlxW.

63. *Мордвинов, Д. А.* Автоматический вывод реляционных инвариантов для нелинейных систем дизъюнктов Хорна с ограничениями [Text]: дис. ... канд. / Мордвинов Дмитрий Александрович. — Санкт-Петербургский государственный университет, 2020.

64. *Itzhaky, S.* Hyperproperty Verification as CHC Satisfiability [Text] / S. Itzhaky, S. Shoham, Y. Vizel // CoRR. — 2023. — Vol. abs/2304.12588. — arXiv: 2304.12588. — URL: https://doi.org/10.48550/arXiv.2304.12588.

65. *Cousot, P.* Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints [Text] / P. Cousot, R. Cousot // Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. — Los Angeles, California: Association for Computing Machinery, 1977. — P. 238–252. — (POPL '77). — URL: https://doi.org/10.1145/512950.512973.

66. *Giacobazzi, R.* Making Abstract Interpretations Complete [Text] / R. Giacobazzi, F. Ranzato, F. Scozzari // J. ACM. — New York, NY, USA, 2000. — Vol. 47, № 2. — P. 361–416. — URL: https://doi.org/10.1145/333979.333989.

67. *Giacobazzi, R.* Analyzing program analyses [Text] / R. Giacobazzi, F. Logozzo, F. Ranzato // ACM SIGPLAN Notices. — 2015. — Vol. 50, № 1. — P. 261–273.

68. *Cousot, P.* Abstract Interpretation Frameworks [Text] / P. Cousot, R. Cousot // Journal of Logic and Computation. — 1992. — Vol. 2, № 4. — P. 511–547. — eprint: https://academic.oup.com/logcom/article-pdf/2/4/511/2740133/2-4-511.pdf. — URL: https://doi.org/10.1093/logcom/2.4.511.

69. *Campion, M.* Partial (In)Completeness in Abstract Interpretation: Limiting the Imprecision in Program Analysis [Text] / M. Campion, M. Dalla Preda, R. Giacobazzi // Proc. ACM Program. Lang. — New York, NY, USA, 2022. — Vol. 6, POPL. — URL: https://doi.org/10.1145/3498721.

70. A Logic for Locally Complete Abstract Interpretations [Text] / R. Bruni [et al.] // 2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). — 2021. — P. 1–13.

71. *Moura, L. de.* Z3: An Efficient SMT Solver [Text] / L. de Moura, N. Bjørner // Tools and Algorithms for the Construction and Analysis of Systems / Ed. by C. R. Ramakrishnan, J. Rehof. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. — P. 337–340.

72. cvc5: A Versatile and Industrial-Strength SMT Solver [Text] / H. Barbosa [et al.] // Tools and Algorithms for the Construction and Analysis of Systems / Ed. by D. Fisman, G. Rosu. — Cham: Springer International Publishing, 2022. — P. 415–442.

73. *Rümmer, P.* A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic [Text] / P. Rümmer // Logic for Programming, Artificial Intelligence, and Reasoning / Ed. by I. Cervesato, H. Veith, A. Voronkov. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. — P. 274–289.

74. *Reger, G.* Instantiation and Pretending to be an SMT Solver with Vampire. [Text] / G. Reger, M. Suda, A. Voronkov // SMT. — 2017. — P. 63–75.

75. *Xi, H.* Dependent Types in Practical Programming [Text] / H. Xi, F. Pfenning // Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — San Antonio, Texas, USA: Association for Computing Machinery, 1999. — P. 214–227. — (POPL '99). — URL: https://doi.org/10.1145/292540.292560.

76. Refinement Types for Haskell [Text] / N. Vazou [et al.] // SIGPLAN Not. — New York, NY, USA, 2014. — Vol. 49, № 9. — P. 269–282. — URL: https://doi.org/10.1145/2692915.2628161.

77. *Unno, H.* Automating Induction for Solving Horn Clauses [Text] / H. Unno, S. Torii, H. Sakamoto // Computer Aided Verification / Ed. by R. Majumdar, V. Kunčak. — Cham: Springer International Publishing, 2017. — P. 571–591.

78. *Hamza, J.* System FR: Formalized Foundations for the Stainless Verifier [Text] / J. Hamza, N. Voirol, V. Kunčak // Proc. ACM Program. Lang. — New York, NY, USA, 2019. — Vol. 3, OOPSLA. — URL: https://doi.org/10.1145/3360592.

79. *Chabin, J.* Visibly pushdown languages and term rewriting [Text] / J. Chabin, P. Réty // International Symposium on Frontiers of Combining Systems. — Springer. 2007. — P. 252–266.

80. *Gouranton, V.* Synchronized tree languages revisited and new applications [Text] / V. Gouranton, P. Réty, H. Seidl // International Conference on Foundations of Software Science and Computation Structures. — Springer. 2001. — P. 214–229.

81. *Limet, S.* Weakly regular relations and applications [Text] / S. Limet, P. Réty, H. Seidl // International Conference on Rewriting Techniques and Applications. — Springer. 2001. — P. 185–200.

82. *Chabin, J.* Synchronized-context free tree-tuple languages [Text]: tech. rep. / J. Chabin, J. Chen, P. Réty; Citeseer. — 2006.

83. *Jacquemard, F.* Rigid tree automata [Text] / F. Jacquemard, F. Klay, C. Vacher // International Conference on Language and Automata Theory and Applications. — Springer. 2009. — P. 446–457.

84. *Engelfriet, J.* Multiple context-free tree grammars and multi-component tree adjoining grammars [Text] / J. Engelfriet, A. Maletti // International Symposium on Fundamentals of Computation Theory. — Springer. 2017. — P. 217–229.

85. *Kozen, D.* Automata and Computability [Text] / D. Kozen. — Springer New York, 2012. — (Undergraduate Texts in Computer Science). — URL: https://books.google.ru/books?id=Vo3fBwAAQBAJ.

86. *McCune, W.* Mace4 Reference Manual and Guide [Text] / W. McCune. — 2003. — URL: https://arxiv.org/abs/cs/0310055.

87. *Torlak, E.* Kodkod: A Relational Model Finder [Text] / E. Torlak, D. Jackson // Tools and Algorithms for the Construction and Analysis of Systems / Ed. by O. Grumberg, M. Huth. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. — P. 632–647.

88. *Claessen, K.* New techniques that improve MACE-style finite model finding [Text] / K. Claessen, N. Sörensson // Proceedings of the CADE-19 Workshop: Model Computation-Principles, Algorithms, Applications. — Citeseer. 2003. — P. 11–27.

89. Finite Model Finding in SMT [Text] / A. Reynolds [et al.] // Computer Aided Verification / Ed. by N. Sharygina, H. Veith. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. — P. 640–655.

90. *Reger, G.* Finding Finite Models in Multi-sorted First-Order Logic [Text] / G. Reger, M. Suda, A. Voronkov // Theory and Applications of Satisfiability Testing – SAT 2016 / Ed. by N. Creignou, D. Le Berre. — Cham: Springer International Publishing, 2016. — P. 323–341.

91. *Lisitsa, A.* Finite Models vs Tree Automata in Safety Verification [Text] / A. Lisitsa // 23rd International Conference on Rewriting Techniques and Applications (RTA'12). Vol. 15 / Ed. by A. Tiwari. — Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012. — P. 225–239. — (Leibniz International Proceedings in Informatics (LIPIcs)). — URL: http://drops.dagstuhl.de/opus/volltexte/2012/3495.

92. *Peltier, N.* Constructing infinite models represented by tree automata [Text] / N. Peltier // Annals of Mathematics and Artificial Intelligence. — 2009. — Vol. 56, № 1. — P. 65–85.

93. *Oppen, D. C.* Reasoning about Recursively Defined Data Structures [Text] / D. C. Oppen // Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. — Tucson, Arizona: Association for Computing Machinery, 1978. — P. 151–157. — (POPL '78). — URL: https://doi.org/10.1145/512760.512776.

94. *Kovács, L.* First-Order Theorem Proving and Vampire [Text] / L. Kovács, A. Voronkov // Computer Aided Verification / Ed. by N. Sharygina, H. Veith. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. — P. 1–35.

95. *Schulz, S.* E - a Brainiac Theorem Prover [Text] / S. Schulz // AI Commun. — NLD, 2002. — Vol. 15, № 2, 3. — P. 111–126.

96. *Cruanes, S.* Superposition with Structural Induction [Text] / S. Cruanes // Frontiers of Combining Systems / Ed. by C. Dixon, M. Finger. — Cham: Springer International Publishing, 2017. — P. 172–188.

97. *Goubault-Larrecq, J.* Towards Producing Formally Checkable Security Proofs, Automatically [Text] / J. Goubault-Larrecq // 2008 21st IEEE Computer Security Foundations Symposium. — 2008. — P. 224–238.

98. Property preserving abstractions for the verification of concurrent systems [Text] / C. Loiseaux [et al.] // Formal methods in system design. — 1995. — Vol. 6. — P. 11–44.

99. Global Guidance for Local Generalization in Model Checking [Text] / H. G. Vediramana Krishnan [et al.] // Computer Aided Verification / Ed. by S. K. Lahiri, C. Wang. — Cham: Springer International Publishing, 2020. — P. 101–125.

100. *Hojjat, H.* Deciding and Interpolating Algebraic Data Types by Reduction [Text] / H. Hojjat, P. Rümmer // 2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC). — 2017. — P. 145–152.

101. *Comon, H.* Equational Formulas with Membership Constraints [Text] / H. Comon, C. Delor // Information and Computation. — 1994. — Vol. 112, № 2. — P. 167–216. — URL: https://www.sciencedirect.com/science/article/pii/S089054018471056X.

102. *Kossak, R.* Undefinability and Absolute Undefinability in Arithmetic [Text] / R. Kossak. — 2023. — arXiv: 2205.06022 [math.LO].

103. *Bar-Hillel, Y.* On formal properties of simple phrase structure grammars [Text] / Y. Bar-Hillel, M. Perles, E. Shamir // STUF - Language Typology and Universals. — 1961. — Vol. 14, № 1–4. — P. 143–172. — URL: https://doi.org/10.1524/stuf.1961.14.14.143.

104. *Zhang, T.* Decision Procedures for Recursive Data Structures with Integer Constraints [Text] / T. Zhang, H. B. Sipma, Z. Manna // Automated Reasoning / Ed. by D. Basin, M. Rusinowitch. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. — P. 152–167.

105. *Caferra, R.* Automated model building [Text]. Vol. 31 / R. Caferra, A. Leitsch, N. Peltier. — Springer Science & Business Media, 2013.

106. *Fermüller, C. G.* Model Representation over Finite and Infinite Signatures [Text] / C. G. Fermüller, R. Pichler // Journal of Logic and Computation. — 2007. — Vol. 17, № 3. — P. 453–477.

107. *Fermüller, C. G.* Model Representation via Contexts and Implicit Generalizations [Text] / C. G. Fermüller, R. Pichler // Automated Deduction – CADE-20 / Ed. by R. Nieuwenhuis. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. — P. 409–423.

108. *Teucke, A.* On the Expressivity and Applicability of Model Representation Formalisms [Text] / A. Teucke, M. Voigt, C. Weidenbach // Frontiers of Combining Systems / Ed. by A. Herzig, A. Popescu. — Cham: Springer International Publishing, 2019. — P. 22–39.

109. *Gramlich, B.* Algorithmic Aspects of Herbrand Models Represented by Ground Atoms with Ground Equations [Text] / B. Gramlich, R. Pichler // Automated Deduction—CADE-18 / Ed. by A. Voronkov. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. — P. 241–259.

110. *Matzinger, R.* On computational representations of Herbrand models [Text] / R. Matzinger // Uwe Egly and Hans Tompits, editors. — 1998. — Vol. 13. — P. 86–95.

111. *Matzinger, R.* Computational representations of models in first-order logic [Text]: PhD thesis / Matzinger Robert. — Technische Universität Wien, Austria, 2000.

112. Handbook of Formal Languages, Vol. 3: Beyond Words [Text] / Ed. by G. Rozenberg, A. Salomaa. — Berlin, Heidelberg: Springer-Verlag, 1997.

113. *Veanes, M.* Symbolic tree automata [Text] / M. Veanes, N. Bjørner // Information Processing Letters. — 2015. — Vol. 115, № 3. — P. 418–424. — URL: https://www.sciencedirect.com/science/article/pii/S0020019014002555.

114. *D'Antoni, L.* Minimization of Symbolic Tree Automata [Text] / L. D'Antoni, M. Veanes // Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science. — New York, NY, USA: Association for Computing Machinery, 2016. — P. 873–882. — (LICS '16). — URL: https://doi.org/10.1145/2933575.2933578.

115. *Barrett, C.* The SMT-LIB Standard: Version 2.6 [Text]: tech. rep. / C. Barrett, P. Fontaine, C. Tinelli; Department of Computer Science, The University of Iowa. — 2017. — Available at http://smtlib.cs.uiowa.edu/.

116. *Reger, G.* The Challenges of Evaluating a New Feature in Vampire. [Text] / G. Reger, M. Suda, A. Voronkov // Vampire Workshop. — 2014. — P. 70–74.

117. *Reynolds, A.* Induction for SMT Solvers [Text] / A. Reynolds, V. Kuncak // Verification, Model Checking, and Abstract Interpretation / Ed. by D. D'Souza, A. Lal, K. G. Larsen. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2015. — P. 80–98.

118. *Yang, W.* Lemma Synthesis for Automating Induction over Algebraic Data Types [Text] / W. Yang, G. Fedyukovich, A. Gupta // Principles and Practice of Constraint Programming / Ed. by T. Schiex, S. de Givry. — Cham: Springer International Publishing, 2019. — P. 600–617.

119. *Clocksin, W. F.* Programming in Prolog [Text] / W. F. Clocksin, C. S. Mellish. — 5th ed. — Berlin: Springer, 2003.

120. Property-Directed Inference of Universal Invariants or Proving Their Absence [Text] / A. Karbyshev [et al.] // J. ACM. — New York, NY, USA, 2017. — Vol. 64, № 1. — URL: https://doi.org/10.1145/3022187.

121. Decidability of Inferring Inductive Invariants [Text] / O. Padon [et al.] // Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — St. Petersburg, FL, USA: Association for Computing Machinery, 2016. — P. 217–231. — (POPL '16). — URL: https://doi.org/10.1145/2837614.2837640.

122. *Bjørner, N. S.* Playing with Quantified Satisfaction. [Text] / N. S. Bjørner, M. Janota // LPAR (short papers). — 2015. — Vol. 35. — P. 15–27.

123. *Losekoot, T.* Automata-Based Verification of Relational Properties of Functions over Algebraic Data Structures [Text] / T. Losekoot, T. Genet, T. Jensen // 8th International Conference on Formal Structures for Computation and Deduction (FSCD 2023). Vol. 260 / Ed. by M. Gaboardi, F. van Raamsdonk. — Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. — 7:1–7:22. — (Leibniz International Proceedings in Informatics (LIPIcs)). — URL: https://drops.dagstuhl.de/opus/volltexte/2023/17991.

124. TIP: Tons of Inductive Problems [Text] / K. Claessen [et al.] // Intelligent Computer Mathematics / Ed. by M. Kerber [et al.]. — Cham: Springer International Publishing, 2015. — P. 333–337.

125. *Stump, A.* StarExec: A Cross-Community Infrastructure for Logic Solving [Text] / A. Stump, G. Sutcliffe, C. Tinelli // Automated Reasoning / Ed. by S. Demri, D. Kapur, C. Weidenbach. — Cham: Springer International Publishing, 2014. — P. 367–373.

126. Transition Power Abstractions for Deep Counterexample Detection [Text] / M. Blicha [et al.] // Tools and Algorithms for the Construction and Analysis of Systems / Ed. by D. Fisman, G. Rosu. — Cham: Springer International Publishing, 2022. — P. 524–542.

127. Predicting Rankings of Software Verification Tools [Text] / M. Czech [et al.] // Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Analytics. — Paderborn, Germany: Association for Computing Machinery, 2017. — P. 23–26. — (SWAN 2017). — URL: https://doi.org/10.1145/3121257.3121262.

Code listing list

Figure list

Table list