Федеральное государственное бюджетное образовательное учреждение
высшего образования «Санкт-Петербургский государственный университет
СПбГУ»

На правах рукописи

Костюков Юрий Олегович

Автоматический вывод индуктивных инвариантов программ с алгебраическими типами данных

Научная специальность
2.3.5. Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание учёной степени
кандидата физико-математических наук

Научный руководитель:
Доктор технических наук, Санкт-Петербургский государственный университет, профессор кафедры системного программирования
Кознов Дмитрий Владимирович

Санкт-Петербург — 2023

Оглавление

Стр.

Стр.

Стр.

Введение

Актуальность темы. Программные системы охватывают всё больше сфер человеческой деятельности, и всё острее стоит вопрос об их корректности. Область формальных методов традиционно занимается вопросами качества программ. С 90-х годов XX века в этой области началась новая страница — появились бинарные диаграммы решений, а затем символьная проверка моделей на основе эффективных SAT-решателей, что позволило верифицировать системы с 10^{120} возможными состояниями [1]. Благодаря SAT-революции всё меньше статических анализаторов создаётся «с нуля», всё чаще они надстраиваются над стеком верификации: SAT-решатели для логики высказываний, построенные на их основе SMT-решатели для теорий логики первого порядка, и далее — Хорн-решатели для вывода индуктивных инвариантов.

Новые подходы к статическому анализу дают индустрии много плодов. Так, например, в 2008 году около трети всех детектируемых ошибок при разработке Windows 7 нашёл инструмент SAGE [2], основанный на символьном исполнении и активно использующий SMT-решатель для проверки достижимости ветвей исполнения программ.

В формальных методах большое значение имеют типы данных, так как для них требуются подходящие формализации, чтобы учитывать их при верификации программ. Однако большинство исследований здесь направлено на поддержку «классических» типов данных, таких как целые числа и массивы. Менее исследованными оказываются новые, набирающие популярность, типы данных, например, *алгебраические типы данных (АТД)*[1]. Последние строятся рекурсивно, при помощи объединения и декартового перемножения типов. Используя АТД, можно описывать односвязные списки, бинарные деревья и другие сложные структуры данных. АТД активно используются в функциональных языках, таких как Haskell и OCaml, являясь альтернативой перечислениям и объединениям в языках C и C++. Алгебраические типы данных всё чаще включают в современные языки программирования, используемые в индустрии, например, в языки Rust и Scala, а также в языки самовыполняющихся контрактов, например, Solidity [3]. Так, например,

[1] В зависимости от подхода их также называют *абстрактными типами данных*, *индуктивными типами данных* и *рекурсивными типами данных*.

Twitter использует язык Scala для большинства своих серверных приложений [4], Dropbox — язык Rust для управления потоками данных [5], и в обоих случая активно используются алгебраические типы данных.

Таким образом становится насущной задача обеспечения корректности программ, использующих АТД. Эта задача может быть формализована, а её решение — частично автоматизировано в рамках дедуктивной верификации на основе логики Флойда-Хоара [6; 7] или уточняющих типов (refinement types) [8], как, например, в системах Flux [9] для языка Rust и Leon [10] для языка Scala. Однако такие подходы требуют от пользователя предоставления *индуктивных инвариантов* для доказательства корректности программы, формулировка которых на практике является крайне трудоёмкой задачей. Системы верификации, основанные на самостоятельных языках программирования и поддерживающие АТД, такие как Dafny [11], Why3 [12], Viper [13], F* [14], сталкиваются с той же проблемой. Также следует отметить, что алгебраические типы данных лежат в основе многочисленных интерактивных систем проверки доказательств (interactive theorem prover, ITP), таких как Coq [15], Idris [16], Agda [17], Lean [18]. Методы автоматизации индукции в таких системах, как правило, ограничены синтаксическим перебором, поэтому в процессе доказательства пользователь вынужден осуществлять трудоёмкую деятельность по формулировке точной индукционной гипотезы, что тождественно проблеме вывода индуктивных инвариантов.

Таким образом эти задачи сводятся к задаче автоматического вывода индуктивных инвариантов программ с алгебраическими типами данных. В общем виде она может быть сформулирована при помощи систем *дизъюнктов Хорна с ограничениями* (constrained Horn clauses, CHCs) — логических формул специального вида, которые позволяют точно моделировать работу программы [19].

Поскольку задача автоматического вывода индуктивных инвариантов сводится к задаче поиска модели для системы дизъюнктов Хорна с ограничениями, инструменты автоматического поиска таких моделей (т. н. «Хорн-решатели») могут быть применены в различных контекстах верификации программ [20; 21]. Так, например, инструмент RustHorn [22] использует Хорн-решатели для верификации Rust-программ, а инструмент SolCMC [23] применяется для верификации самовыполняющихся контрактов на языке Solidity.

Существуют эффективные Хорн-решатели, поддерживающие АТД, такие как Spacer [24] и его приемник Racer [25], а также Eldarica [26], HoIce [27],

RCHC [28], VeriCaT [29]. Среди Хорн-решателей проводятся ежегодные международные соревнования CHC-COMP [30], где отдельная секция посвящена решению систем дизъюнктов Хорна с алгебраическими типами данных.

Решение выполнимой системы дизъюнктов Хорна классически представляется в виде т. н. *символьной модели* (symbolic model) [21], т. е. модели, выраженной при помощи формул логики первого порядка в языке ограничений системы дизъюнктов. Поэтому класс всех индуктивных инвариантов, выразимых с помощью языка ограничений, будем называем *классическими символьными инвариантами*. Так, например, все Хорн-решатели, участвовавшие в соревнованиях CHC-COMP за последние два года, строят классические символьные инварианты.

Проблема символьных инвариантов в контексте алгебраических типов данных заключается в том, что язык ограничений АТД *не позволяет выразить индуктивные инварианты большинства программ, востребованных на практике*. А если у безопасной программы нет индуктивных инвариантов, выразимых на языке ограничений, ни один алгоритм вывода индуктивных инвариантов на этом языке не сможет построить для неё индуктивный инвариант. Это приводит к тому, что *Хорн-решатели, строящие классические символьные инварианты, не завершаются на большинстве систем с алгебраическими типами данных*.

Термы алгебраических типов имеют *рекурсивную структуру*. Например, бинарное дерево — это либо лист, либо вершина с двумя потомками, которые тоже являются бинарными деревьями. Поэтому основная причина, по которой язык ограничений АТД не позволяет выразить индуктивные инварианты многих программ, состоит в том, что он не позволяет выражать *рекурсивные отношения* над термами алгебраических типов.

Степень разработанности темы. Проблема невыразимости языка ограничений хорошо известна в научном сообществе. Предпринималось несколько попыток решить эту проблему.

Так в 2018 году Ф. Рюммер (P. Ruemmer, Швеция) в рамках Хорн-решателя Eldarica [26] предложил выводить индуктивные инварианты в языке ограничений, расширенном функцией размера, подсчитывающей число конструкторов в терме. Однако проблемой этого подхода является то, что любое расширение языка ограничений требует существенной переработки всей процедуры вывода индуктивных инвариантов. Также в 2022 году в рамках

Хорн-решателя Racer (H. Govind, A. Gurfinkel, США) [25] было предложено расширить язык ограничений катаморфизмами — рекурсивными функциями некоторого простого вида. Однако в этом случае от пользователя требуется заранее задавать катаморфизмы, которые будут использованы для построения индуктивного инварианта, поэтому этот подход нельзя назвать вполне автоматическим.

С 2018 года ведётся отдельная линия исследований (E. De Angelis, F. Fioravant, A. Pettorossi, Италия) [31—34], посвящённая методам устранения алгебраических типов из системы дизъюнктов путём сведения её к системе над более простой теорией, например, над линейной арифметикой. Такой подход реализован в инструменте VeriCaT [29]. Ограничением подобных подходов является невозможность восстановления индуктивного инварианта исходной системы из индуктивного инварианта более простой системы.

В 2020 году в рамках инструмента RCHC (T. Haudebourg, Франция) [28] было предложено выражать индуктивные инварианты программ над АТД при помощи *автоматов над деревьями* [35]. Однако из-за сложностей с представлением кортежей термов автоматами, предложенный класс инвариантов не включает в себя классические символьные инварианты, поэтому подход часто оказывается неприменимым для простейших программ, где такие инварианты легко находятся классическими методами.

Целью данной работы является предложение новых классов индуктивных инвариантов для программ с алгебраическими типами данных и создание для них методов автоматического вывода. Для реализации этой цели были сформулированы следующие **задачи**.

1. Предложить новые классы индуктивных инвариантов программ с алгебраическими типами данных, позволяющие выражать рекурсивные отношения и включающие классические символьные инварианты.
2. Создать методы автоматического вывода индуктивных инвариантов в новых классах.
3. Выполнить пилотную программную реализацию предложенных методов.
4. Провести экспериментальное сопоставление реализованного инструмента с существующими на представительном тестовом наборе.

Методология и методы исследования. Методология исследования заключается в проектировании применимых на практике классов индуктивных

инвариантов совместно с разработкой соответствующих алгоритмов, активно используя существующие результаты этой области. В работе используется логика первого порядка, а также базовые концепции теории автоматов и формальных языков, включая автоматы над деревьями, синхронные автоматы, язык автомата, лемму о «накачке». Пилотная программная реализация теоретических результатов выполнена на языке F#, а также частично на языке C++ в рамках кодовой базы Хорн-решателя RACER (входит в SMT-решатель Z3, Microsoft Research).

Основные положения, выносимые на защиту.

1. Предложен эффективный метод автоматического вывода индуктивных инвариантов, основанных на автоматах над деревьями; при этом данные инварианты позволяют выражать рекурсивные отношения в большем количестве реальных программ; метод базируется на поиске конечных моделей.
2. Предложен метод автоматического вывода индуктивных инвариантов, основанный на трансформации программы и поиске конечных моделей, в сложном для автоматического вывода инвариантов классе, основанном на синхронных автоматах над деревьями; этот класс инвариантов позволяет выражать рекурсивные отношения и обобщает классические символьные инварианты.
3. Предложен класс индуктивных инвариантов, основанный на булевой комбинации классических инвариантов и автоматов над деревьями, который, с одной стороны, позволяет выражать рекурсивные отношения в реальных программах, а, с другой стороны, позволяет эффективно выводить индуктивные инварианты; также предложен эффективный метод совместного вывода индуктивных инвариантов в этом классе посредством вывода инвариантов в подклассах.
4. Проведено теоретическое сравнение существующих и предложенных в рамках диссертации классов индуктивных инвариантов; в том числе сформулированы и доказаны леммы о «накачке» для языка ограничений и для языка ограничений расширенного функцией размера терма.
5. Выполнена пилотная программная реализация предложенных методов на языке F# в рамках инструмента RINGEN; разработанный инструмент сопоставлен с существующими методами на общепринятом тестовом наборе задач верификации функциональных программ «Tons

of Inductive Problems»; реализация наилучшего из предложенных методов смогла за отведённое время решить в 3.74 раза больше задач, чем наилучший из существующих инструментов.

Научная новизна полученных результатов состоит в следующем.

1. Впервые предложен класс индуктивных инвариантов, основанный на булевой комбинации классов классических и инвариантов, основанных на автоматах над деревьями.
2. Впервые предложен алгоритм вывода индуктивных инвариантов для программ с алгебраическими типами данных, основанный на поиске конечных моделей.
3. Предложен новый алгоритм совместного вывода индуктивных инвариантов в комбинации классов инвариантов на базе имеющихся методов вывода инвариантов для отдельных классов.
4. Впервые введены и доказаны леммы о «накачке» для языков первого порядка в сигнатуре теории алгебраических типов данных.

Теоретическая значимость работы. Диссертационное исследование предлагает новые подходы к выводу индуктивных инвариантов программ. Поскольку эти подходы ортогональны существующим, они могут быть перенесены на программы над другими теориями, например, над теорией массивов, а также могут усилить уже существующие подходы к выводу индуктивных инвариантов. Также важным теоретическим вкладом является адаптация лемм о «накачке» к языкам первого порядка: эти леммы открывают путь к фундаментальному исследованию проблемы невыразимости индуктивных инвариантов в языках первого порядка и проектированию новых классов индуктивных инвариантов программ.

Практическая значимость работы. Предложенные методы могут быть применены при создании статических анализаторов для языков с алгебраическими типами данных: поскольку индуктивные инварианты аппроксимируют циклы и функции, они позволяют анализатору корректно «срезать» целые классы недостижимых состояний программы и не «увязать» в циклах и рекурсии. Например, предложенные методы могут быть полезны в разработке верификаторов и генераторов тестовых покрытий для таких языков, как Rust, Scala, Solidity, Haskell и OCaml. Поскольку для предложенных методов была выполнена пилотная программная реализация, полученный

Хорн-решатель также может быть использован в качестве «ядра» статического анализатора, например, для языка Rust при помощи фреймворка RustHorn.

Достоверность полученных результатов обеспечивается формальными доказательствами, а также компьютерными экспериментами на публичных общепринятых тестовых наборах. Полученные в диссертации результаты согласуются с результатами других авторов в области вывода индуктивных инвариантов.

Апробация работы. Основные результаты работы докладывались на следующих научных конференциях и семинарах: международном семинаре HCVS 2021 (28 марта 2021, Люксембург), семинаре компании Huawei (18-19 ноября 2021, Санкт-Петербург), ежегодном внутреннем семинаре JetBrains Research (18 декабря 2021, Санкт-Петербург), конференции PLDI 2021 (23-25 июня 2021, Канада), внутреннем семинаре Венского технического университета (3 июня 2022, Австрия), конференции LPAR 2023 (4-9 июня 2023, Колумбия).

Разработанный инструмент в 2021 и 2022 годах занял, соответственно, 2 и 1 место на международных соревнованиях CHC-COMP (секция по выводу индуктивных инвариантов для программ с алгебраическими типами данных).

Публикации. Основные результаты по теме диссертации изложены в 4 печатных изданиях, 2 из которых изданы в журналах, рекомендованных ВАК, 2 — в периодических научных журналах, индексируемых Web of Science и Scopus, одна из которых опубликована в тезисах конференции PLDI, имеющей ранг A*, и одна опубликована в тезисах конференции LPAR, имеющей ранг A.

Личный вклад автора в совместных публикациях распределён следующим образом. В статье [36] автор выполнил реализацию сведения поиска индуктивных инвариантов функций над сложными структурами данных к решению систем дизъюнктов Хорна, а также спроектировал эксперименты с различными существующими Хорн-решателями; соавторы предложили саму идею и проработали её теоретические аспекты. В работах [37] автор провёл теоретическое сопоставление классов индуктивных инвариантов, предложил и доказал леммы о «накачке» для языков первого порядка над АТД, реализовал предлагаемый подход, поставил эксперименты; соавторы участвовали в обсуждении основных идей статьи, выполнили обзор существующих решений. В статье [38] автор предложил и формально обосновал коллаборационный подход к выводу инвариантов, реализовал прототип и поставил эксперименты; соавторы участвовали в обсуждении презентации идей статьи и выполнили обзор

существующих решений. В статье [39] вклад автора заключается в формальном описании теории вычисления предусловий программ со сложными структурами данных; соавторы участвовали в обсуждении основных идей и реализовали подход.

Объём и структура работы. Диссертация состоит из введения, 6 глав и заключения. Полный объём диссертации составляет 110 страниц, включая 5 листингов кода, 6 рисунков и 4 таблицы. Список литературы содержит 130 наименований.

Глава 1. Обзор предметной области

В данной главе представлены ключевые для данного диссертационного исследования понятия и теоремы, а также описано состояние предметной области на момент написания работы. В разделе 1.2 приведена краткая история проблемы выразимости индуктивных инвариантов, которая является базовой для данного диссертационного исследования. В разделе 1.3 формально определены язык ограничений, логика первого порядка и алгебраические типы данных; именно с этими объектами будут оперировать предложенные в данной работе методы верификации. В разделе 1.4 представлены системы дизъюнктов Хорна и показана их связь с задачей верификации программ. Для описания множеств термов алгебраических типов данных в разделе 1.5 приведены базовые понятия формальных языков деревьев. Наконец, в разделе 1.6 представлены выводы по обзору.

1.1 Краткая история верификации программ

Историю верификации принято начинать с отрицательных результатов: проблемы останова Тьюринга (1936 г.) [40] и теоремы Райса (1953 г.) [41], которые говорят о невозможности существования верификатора, останавливающегося на всех входах и дающего только корректные результаты. Первые конструктивные попытки создать подходы для верификации программ были предприняты Р. В. Флойдом (1967 г.) [6] и Ч. Э. Р. Хоаром (1969 г.) [7]. Эти исследователи развили методы, основывающиеся на сведении верификации к проверке логических условий. Первым практичным подходом к верификации считается *проверка моделей* (model checking, 1981 г.) [42], возникшая в контексте верификации конкурентных программ. Её существенным ограничением был т. н. «взрыв пространства состояний» [43]: пространство состояний растёт *экспоненциально* с ростом размерности состояния.

Для решения этой проблемы К. Макмилланом была предложена *символьная проверка моделей* (1987 г.), реализованная в инструменте SMV (1993 г.) [1]. С 1996 года произошёл переход к представлению множеств состояний программы SAT-формулами логики высказываний (SATisfiability) [44], что позволило

верифицировать системы, содержащие до 10^{120} состояний [1]. Это стало возможным благодаря новому поколению SAT-решателей, таких как Chaff [45], основанных на алгоритме проверки выполнимости формул с нехронологическим возвратом — CDCL (conflict driven clause learning) [46]. На основе CDCL в 2002 г. был предложен алгоритм CDCL(T) для проверки выполнимости формул логики первого порядка в разных теориях (satisfiability modulo theories, SMT) [47], спроектированных специально для задач в области формальных методов. В 2002 г. был реализован первый SMT-решатель CVC [48], использующий SAT-решатель Chaff.

Появление эффективных SAT и SMT-решателей позволило вынести из процесса верификации проверку логических условий. В 1999 г. была предложена *ограничиваемая проверка моделей* (bounded model checking, BMC) [49], строящая логические формулы из раскруток отношения переходов программы и отдающая их в сторонний решатель. Затем, в 1995—2000 гг., благодаря Р. П. Куршану и Э. Кларку, появился метод *направляемого контрпримерами уточнения абстракций* (counterexample-guided abstraction refinement, CEGAR) [50; 51], который позволил верифицировать программы путём итеративного построения индуктивных инвариантов в виде абстракций и их уточнения при помощи контрпримеров к индуктивности инвариантов программ. В 2003—2005 гг. К. Макмилланом было предложено строить абстракции при помощи *интерполянтов* невыполнимых формул, извлекаемых из логического решателя [52; 53]. Интерполянты при этом, по сути, являются локальными (частичными) доказательствами корректности программы.

В 2012 г. было предложено внедрить в стек «верификатор, SMT-решатель, SAT-решатель» еще также *Хорн-решатель*, отвечающего за автоматический вывод индуктивных инвариантов и контрпримеров к спецификации [20]. Тем самым роль верификатора свелась к синтаксической редукции программы к системе дизъюнктов Хорна, а «ядром» процесса верификации становится Хорн-решатель. Так, например, CEGAR был реализован в Хорн-решателе Eldarica. В 2014 г. П. Гаргом был предложен подход ICE, основанный на обучении с учителем [54]. ICE реализован, например, в Хорн-решателях HoIce и RCHC.

В 2011 г. А. Р. Брэдли был предложен подход IC3/PDR [55] для верификации аппаратного обеспечения на основе SAT-решателей. К 2014 г. подход был обобщён для верификации программного обеспечения на основе SMT-решателей [56; 57]. IC3/PDR усиливает CEGAR, создавая абстракции путём

построения индуктивных усилений спецификации, при этом равномерно распределяя ресурсы между поиском индуктивного инварианта и контрпримера. IC3/PDR реализован в Хорн-решателях Spacer [24] и Racer [25].

Благодаря эффективным алгоритмам Хорн-решатели всё больше применяются при верификации реальных программ, например, самоисполняющихся контрактов.

1.2 История проблемы выразимости индуктивных инвариантов

После появления логики Флойда-Хоара в 1967−1969 гг. [6; 7] остро встал вопрос о достаточности предложенного исчисления для доказательства корректности всех возможных программ. Иными словами, сразу была доказана корректность исчисления, но на долгие годы оставалась нерешённой проблема его *полноты*, т. е. что предложенного исчисления достаточно, чтобы доказать безопасность всех безопасных программ. Занимаясь это проблемой, в 1978 г. С. А. Кук доказал [58] *относительную* полноту логики Хоара. Ограничение относительной полноты в теореме состояло в том, что все возможные слабейшие предусловия должны быть выразимы в языке ограничений. Примерно с этого времени стали накапливались примеры простых программ, чьи инварианты невыразимы в языке ограничений [59]. Поэтому в 1987 г. А. Бласс и Ю. Гуревич предложили отказаться от логики первого порядка в пользу *экзистенциальной логики с неподвижной точкой* (existential fixed-point logic) [60; 61]. Она существенно более выразительна, чем логика первого порядка, поэтому для неё была доказана классическая теорема о полноте.

Следует отметить, что формулы экзистенциальной логики с неподвижной точкой соответствуют (при взятии отрицания) системам дизъюнктов Хорна с ограничениями [21]. А последние позволяют выразить все возможные индуктивные инварианты программ, однако они не являются *эффективным* представлением: задача проверки выполнимости систем дизъюнктов Хорна в общем случае неразрешима. Поэтому проблема выразимости инвариантов не исчезла, но трансформировалась в основную проблему данного диссертационного исследования: *как выражать и эффективно строить решения систем дизъюнктов Хорна с ограничениями?*

На данный момент предлагаются различные подходы к практическому решению этой проблемы: от трансформации систем дизъюнктов в системы, у

которых существование выразимого инварианта более вероятно (см. работы 2015–2022 гг. E. De Angelis, F. Fioravant, A. Pettorossi [31–34; 62; 63]), в т. ч. синтаксических синхронизаций дизъюнктов [63; 64], до вывода реляционных инвариантов (инвариантов для нескольких предикатов) [65; 66].

Фактически, решению того же вопроса посвящены исследования в области *полноты абстрактной интерпретации* — возникшего в 1977 г. подхода [67], который позволяет строить статические анализаторы, корректные по построению. Неполнота возникает из-за аппроксимации неразрешимых свойств в разрешимом абстрактном домене, например, в разрешимом фрагменте логики первого порядка.

В 2000 г. было показано, что абстрактный домен можно уточнять по мере работы анализатора [68]. Однако, как в 2015 г. показали Р. Джакобацци и др. [69], это может привести к слишком точному абстрактному домену, в результате чего анализатор не будет завершаться. Поэтому важнейшей частью проектирования абстрактного интерпретатора является построение абстрактного домена под конкретный класс задач [70]. Последние работы в области [71; 72] направлены на исследование точности анализа и *локальной полноты*: полноты относительно заданного набора трасс.

1.3 Язык ограничений

Для произвольного множества X определим следующие множества: $X^n \triangleq \{\langle x_1, \ldots, x_n\rangle \mid x_i \in X\}$ и $X^{\leqslant n} \triangleq \bigcup_{i=1}^{n} X^i$.

1.3.1 Синтаксис и семантика языка ограничений

Многосортная сигнатура первого порядка с равенством является кортежем $\Sigma = \langle \Sigma_S, \Sigma_F, \Sigma_P\rangle$, где Σ_S — множество сортов, Σ_F — множество функциональных символов, Σ_P — множество предикатных символов, среди которых есть выделенный символ равенства $=_\sigma$ для каждого сорта σ (индекс сорта у равенства везде далее будет опущен). Каждый функциональный символ $f \in \Sigma_F$ имеет арность $\sigma_1 \times \cdots \times \sigma_n \to \sigma$, где $\sigma_1, \ldots, \sigma_n, \sigma \in \Sigma_S$, а каждый предикатный символ $p \in \Sigma_P$ имеет арность $\sigma_1 \times \cdots \times \sigma_n$. Термы, атомы, формулы, замкнутые формулы и предложения языка первого порядка (ЯПП) определяются также, как обычно. Язык первого порядка, определённый над сигнатурой Σ, будет называться *языком ограничений*, а формулы в нём — Σ-формулами.

Многосортная структура (модель) $\mathcal{M}$ для сигнатуры Σ состоит из непустых носителей $|\mathcal{M}|_\sigma$ для каждого сорта $\sigma \in \Sigma_S$. Каждому функциональному символу f с арностью $\sigma_1 \times \cdots \times \sigma_n \to \sigma$ сопоставим интерпретацию $\mathcal{M}[\![f]\!] : |\mathcal{M}|_{\sigma_1} \times \cdots \times |\mathcal{M}|_{\sigma_n} \to |\mathcal{M}|_\sigma$, и каждому предикатному символу p с арностью $\sigma_1 \times \cdots \times \sigma_n$ сопоставим интерпретацию $\mathcal{M}[\![p]\!] \subseteq |\mathcal{M}|_{\sigma_1} \times \cdots \times |\mathcal{M}|_{\sigma_n}$. Для каждого замкнутого терма t с сортом σ интерпретация $\mathcal{M}[\![t]\!] \in |\mathcal{M}|_\sigma$ определяется рекурсивно естественным образом.

Структура называется конечной, если все её носители всех сортов конечны, в противном случае она называется бесконечной.

Выполнимость предложения φ в модели $\mathcal{M}$ обозначается $\mathcal{M} \models \varphi$ и определяется, как обычно. Употреблением $\varphi(x_1, \ldots, x_n)$ вместо φ будет подчёркиваться, что все свободные переменные в φ находятся среди $\{x_1, \ldots, x_n\}$. Далее, $\mathcal{M} \models \varphi(a_1, \ldots, a_n)$ обозначает, что $\mathcal{M}$ выполняет φ на оценке, сопоставляющей свободным переменным элементы соответствующих носителей $a_1, \ldots, a_n$ (переменные также связаны с сортами). Универсальное замыкание формулы $\varphi(x_1, \ldots, x_n)$ обозначается $\forall\varphi$ и определяется как $\forall x_1 \ldots \forall x_n.\varphi$. Если φ имеет свободные переменные, то $\mathcal{M} \models \varphi$ означает $\mathcal{M} \models \forall\varphi$. Формула называется *выполнимой в свободной теории*, если она выполнима в некоторой модели той же сигнатуры.

1.3.2 Алгебраические типы данных

Алгебраический тип данных (АТД) является кортежем $\langle C, \sigma\rangle$, где σ — это сорт данного АТД, C — множество функциональных символов-конструкторов. В научной литературе АТД также называют *абстрактными типами данных*, *индуктивными типами данных* и *рекурсивными типами данных*. Они позволяют задавать такие структуры данных как списки, бинарные и красно-чёрные деревья и др.

Пусть дан набор АТД $\langle C_1, \sigma_1\rangle, \ldots, \langle C_n, \sigma_n\rangle$ такой, что $\sigma_i \neq \sigma_j$ и $C_i \cap C_j = \varnothing$ при $i \neq j$. В связи с фокусом данной работы далее мы будет рассматривать только сигнатуры теории алгебраических типов данных $\Sigma = \langle \Sigma_S, \Sigma_F, \Sigma_P\rangle$, где $\Sigma_S = \{\sigma_1, \ldots, \sigma_n\}$, $\Sigma_F = C_1 \cup \cdots \cup C_n$ и $\Sigma_P = \{=_{\sigma_1}, \ldots, =_{\sigma_n}\}$. Поскольку Σ не имеет предикатных символов, отличных от символов равенства (которые имеют фиксированные интерпретации внутри каждой структуры), существует единственная эрбрановская модель $\mathcal{H}$ для Σ. Носитель эрбрановской модели $\mathcal{H}$ — это кортеж $\langle |\mathcal{H}|_{\sigma_1}, \ldots, |\mathcal{H}|_{\sigma_n}\rangle$, где каждое множество $|\mathcal{H}|_{\sigma_i}$ — это все замкнутые

термы сорта σ_i. Эрбрановская модель интерпретирует все замкнутые термы ими самими, поэтому служит стандартной моделью для теории алгебраических типов данных. Формула φ языка ограничений будет называться *выполнимой по модулю теории* АТД, если имеем $\mathcal{H} \models \varphi$.

Выполнимость формул в свободной теории, а также в теории АТД, может быть проверена автоматически при помощи так называемых *SMT-решателей*, таких как Z3 [73], CVC5 [74] и PRINCESS [75], и посредством автоматических инструментов доказательства теорем, таких как VAMPIRE [76]. Эти инструменты позволяют отделить задачу поиска доказательства безопасности программы от проверки таких доказательств, автоматизируя последнюю задачу.

1.4 Системы дизъюнктов Хорна с ограничениями

Системы дизъюнктов Хорна — логический способ представлять программы совместно с их спецификациями. К задаче проверки выполнимости систем дизъюнктов Хорна сводится задача верификации программ на самых разных языках программирования, от функциональных до объектно-ориентированных [21]. Поэтому задача вывода индуктивных инвариантов в данной работе ставится и исследуется в формулировке для систем дизъюнктов Хорна, а дизъюнкты Хорна являются ключевым понятием для всей дальнейшей работы.

1.4.1 Синтаксис

Пусть $\mathcal{R} = \{P_1, \ldots, P_n\}$ является конечным множеством предикатных символов с сортами из сигнатуры Σ. Такие символы будут называться *неинтерпретированными*. Формула C над сигнатурой $\Sigma \cup \mathcal{R}$ называется *дизъюнктом Хорна с ограничениями*, если эта формула имеет следующий вид:

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

Здесь φ — это *ограничение* (формула языка ограничений без кванторов), $R_i \in \mathcal{R}$, а $\bar{t}_i$ — кортеж термов. H называется *головой* дизъюнкта и может быть либо ложью $\bot$ (тогда дизъюнкт называется *запросом*), либо атомарной формулой $R(\bar{t})$ (тогда дизъюнкт называется *правилом для* R). При этом $R \in \mathcal{R}$ и $\bar{t}$ является кортежем термов. Множество всех правил для $R \in \mathcal{R}$ обозначается $rules\,(R)$. Посылка импликации $\varphi \wedge R_1(\bar{t}_1) \wedge \ldots \wedge R_m(\bar{t}_m)$ называется *телом* формулы C и обозначается $body\,(C)$.

Системой дизъюнктов Хорна $\mathcal{P}$ называется конечное множество дизъюнктов Хорна с ограничениями.

1.4.2 Выполнимость и безопасные индуктивные инварианты

Пусть $\overline{X} = \langle X_1, \ldots, X_n \rangle$ является кортежем таких отношений, что если предикат P_i имеет сорт $\sigma_1 \times \ldots \times \sigma_m$, то справедливо $X_i \subseteq |\mathcal{H}|_{\sigma_1} \times \ldots \times |\mathcal{H}|_{\sigma_m}$. Для упрощения обозначений расширение модели $\mathcal{H}\{P_1 \mapsto X_1, \ldots, P_n \mapsto X_n\}$ будет записываться как $\langle \mathcal{H}, X_1, \ldots, X_n \rangle$ или просто $\langle \mathcal{H}, \overline{X} \rangle$.

Система дизъюнктов Хорна $\mathcal{P}$ называется *выполнимой по модулю теории* АТД (или *безопасной*), если существует кортеж отношений $\overline{X}$ такой, что выполнено $\langle \mathcal{H}, \overline{X} \rangle \models C$ для всех формул $C \in \mathcal{P}$. В таком случае кортеж $\overline{X}$ называется *(безопасным индуктивным) инвариантом* системы $\mathcal{P}$. Таким образом, по определению система дизъюнктов Хорна выполнима тогда и только тогда, когда у неё существует безопасный индуктивный инвариант.

Поскольку индуктивный инвариант $\overline{X}$ — это кортеж множеств, которые для большинства систем будут бесконечными, то для автоматического вывода индуктивных инвариантов выбирается некоторый фиксированный класс, элементы которого выразимы некоторым конечным образом. Именно такие классы рассматриваются в данной работе.

Важно отметить, что у системы дизъюнктов Хорна может не быть индуктивного инварианта (если она невыполнима), может быть один индуктивный инвариант, а также их может быть несколько, в том числе бесконечно много. Также важно, что если некоторый алгоритм ищет индуктивные инварианты системы дизъюнктов Хорна только в заранее заданном классе, то может возникнуть ситуация, что данная система выполнима, однако ни один из её индуктивных инвариантов невыразим в этом классе. Как правило, это приводит к тому, что на такой системе алгоритм не завершает свою работу.

Здесь и далее нотация $\mathcal{P} \in \mathcal{C}$, где $\mathcal{P}$ — название примера с некоторой системой дизъюнктов Хорна *с одним неинтерпретированным символом*, а $\mathcal{C}$ — некоторый класс индуктивных инвариантов, означает, что система $\mathcal{P}$ безопасна и *некоторый* её безопасный индуктивный инвариант (отношение, интерпретирующее единственный предикат) лежит в классе $\mathcal{C}$.

Определение 1 (Elem). Отношение $X \subseteq |\mathcal{M}|_{\sigma_1} \times \cdots \times |\mathcal{M}|_{\sigma_n}$ *выразимо языком АТД первого порядка* (*элементарно*), если существует Σ-формула

$\varphi(x_1, \ldots, x_n)$ такая, что $(a_1, \ldots, a_n) \in X$ тогда и только тогда, когда $\mathcal{H} \models \varphi(a_1, \ldots, a_n)$. Класс всех элементарных отношений будем обозначать Elem. Инварианты, лежащие в этом классе, называются *элементарными*, а также *классическими символьными инвариантами.*

Элементарные инварианты с ограничениями размера термов

Инструмент Eldarica [26] выводит инварианты системы дизъюнктов Хорна над АТД в расширении языка ограничений ограничениями на размер термов. Определим класс инвариантов, выразимых формулами этого языка.

Определение 2 (SizeElem). Сигнатура SizeElem получается из языка Elem путем добавления в сорта Int, операций из арифметики Пресбургера и функциональных символов $size_\sigma$ с арностью $\sigma \to Int$. Для краткости мы будем опускать знак σ в символах $size$.

Выполнимость формул с ограничениями на размер термов проверяется в структуре $\mathcal{H}_{size}$, полученной путём соединения стандартной модели арифметики Пресбургера с эрбрановской моделью $\mathcal{H}$ и следующей интерпретацией функции размера:

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

Например, размер терма $t \equiv cons\big(Z, cons(S(Z), nil)\big)$ в объединённой структуре вычисляется следующим образом: $\mathcal{H}_{size}[\![size(t)]\!] = 6$.

1.4.3 Невыполнимость и резолютивные опровержения

Хорошо известно, что невыполнимость системы дизъюнктов Хорна может быть засвидетельствована резолютивным опровержением.

Определение 3. *Резолютивное опровержение* (дерево опровержений) системы дизъюнктов Хорна $\mathcal{P}$ — это конечное дерево с вершинами $\langle C, \Phi \rangle$, где

(1) $C \in \mathcal{P}$ и Φ — это $\Sigma \cup \mathcal{R}$-формула;
(2) в корне дерева находится запрос C и выполнимая Σ-формула Φ;
(3) в каждом листе дерева содержится пара $\langle C, body\,(C) \rangle$, причём $body\,(C)$ является Σ-формулой;
(4) вершина дерева $\langle C, \Phi \rangle$ имеет детей $\langle C_1, \Phi_1 \rangle, \ldots, \langle C_n, \Phi_n \rangle$, если верно следующее:

– $body\,(C) \equiv \varphi \wedge P_1(\overline{x}_1) \wedge \ldots \wedge P_n(\overline{x}_n)$;
– $C_i \in rules\,(P_i)$;
– $\Phi \equiv \varphi \wedge \Phi_1(\overline{x}_1) \wedge \ldots \wedge \Phi_n(\overline{x}_n)$.

Теорема 1. У системы дизъюнктов Хорна есть резолютивное опровержение тогда и только тогда, когда она невыполнима.

1.4.4 От верификации к решению систем дизъюнктов Хорна

Инструменты, позволяющие автоматически решать задачу проверки выполнимости системы дизъюнктов Хорна, называются *Хорн-решателями*. Как правило, Хорн-решатель для некоторой системы дизъюнктов возвращает индуктивный инвариант или резолютивное опровержение, хотя также может вернуть результат «неизвестно» или не завершиться.

При помощи различных теоретических подходов задача верификации программ может быть сведена к задаче проверки выполнимости системы дизъюнктов Хорна [20; 21]. Среди таких теоретических подходов значимыми являются логика Флойда-Хоара для императивных программ [6; 7], а также зависимые (dependent types) [77] и уточняющие типы (refinement types) [8] для функциональных программ. Существует множество инструментов, в рамках которых может быть реализовано это сведение, например, LiquidHaskell [78] для языка Haskell, RCaml [79] для языка OCaml, Flux [9] для языка Rust, Leon [10] и Stainless [80] для языка Scala. На основе этих подходов построены такие инструменты, как RustHorn [22] — верификатор для языка Rust, и SolCMC [23] — верификатор самовыполняющихся контрактов на языке Solidity. Эти инструменты напрямую используют Хорн-решатели с поддержкой АТД, такие как Spacer и Eldarica.

1.5 Языки деревьев

Различные множества АТД-термов, называемые языками деревьями, исследуются в рамках формальных языков как обобщения языков строк. В частности, исследуется обобщение (строковых) автоматов до автоматов над деревьями, а также различные их расширения, как правило, обладающие свойствами разрешимости и замкнутости базовых языковых операций (например, проверки на пустоту пересечения языков) [81—86]. Для данной работы различные классы языков деревьев представляют интерес, поскольку они могут

служить в качестве классов безопасных индуктивных инвариантов программ, использующих АТД.

1.5.1 Свойства и операции

Для построения эффективного алгоритма вывода инвариантов от класса инвариантов, как правило, требуются следующие свойства: замкнутость относительно булевых операций, разрешимость задачи принадлежности кортежа инварианту и разрешимость задачи проверки пустоты инварианта.

Определение 4 (Замкнутость). Пусть операция $\bowtie$ — это или $\cap$ (пересечение множеств), или $\cup$ (объединение множеств), или $\setminus$ (вычитание множеств). Класс множеств называется *замкнутым относительно бинарной операции* $\bowtie$, если для каждой пары множеств X и Y из данного класса множество $X \bowtie Y$ также лежит в этом классе.

Определение 5 (Разрешимость принадлежности). Задача по определению принадлежности кортежа замкнутых термов некоторому множеству термов разрешима в некотором классе множеств термов тогда и только тогда, когда разрешимо множество пар из кортежей замкнутых термов $\bar{t}$ и элементов этого класса i таких, что i выражает некоторое множество I и выполняется $\bar{t} \in I$.

Определение 6 (Разрешимость пустоты). Задача определения пустоты множества разрешима в классе множеств термов тогда и только тогда, когда разрешимо множество элементов класса, выражающих пустое множество.

1.5.2 Автомат над деревьями

Автоматы над деревьями являются обобщением классических строковых автоматов на языки деревьев (языки термов), сохраняющим свойства разрешимости и замкнутости базовых операций. Классические результаты для автоматов над деревьями и их расширений представлены в книге [35].

Определение 7. *(Конечный) n-автомат (над деревьями)* над алфавитом Σ_F является кортежем $\langle S, \Sigma_F, S_F, \Delta \rangle$, где S — это (конечное) множество состояний, $S_F \subseteq S^n$ — множество конечных состояний, Δ — отношение перехода с

правилами следующего вида:

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

Здесь использованы следующие обозначения: функциональные символы — $f \in \Sigma_F$, их арность — $ar(f) = m$ и состояния — $s, s_1, \ldots, s_m \in S$.

Автомат называется *детерминированным*, если в Δ нет правил с совпадающей левой частью.

Определение 8. Кортеж замкнутых термов $\langle t_1, \ldots, t_n \rangle$ *принимается* (допускается) n-автоматом $A = \langle S, \Sigma_F, S_F, \Delta \rangle$, если $\langle A[t_1], \ldots, A[t_n] \rangle \in S_F$, где

$$A\big[f(t_1, \ldots, t_m)\big] \triangleq \begin{cases} s, & \text{если } \big(f(A[t_1], \ldots, A[t_m]) \to s\big) \in \Delta, \\ \text{не определено}, & \text{иначе.} \end{cases}$$

Язык автомата A, обозначаемый $\mathcal{L}(A)$, — это множество всех допустимых автоматом A кортежей термов.

Пример 1. Пусть $\Sigma = \langle Prop, \{(\_ \wedge \_), (\_ \to \_), \top, \bot\}, \varnothing \rangle$ является сигнатурой логики высказываний. Рассмотрим автомат $A = \langle \{q_0, q_1\}, \Sigma_F, \{q_1\}, \Delta \rangle$ с набором отношений перехода Δ, представленными ниже.

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

Автомат A допускает только истинные пропозициональные формулы без переменных.

1.5.3 Конечные модели

Существует взаимно-однозначное соответствие между конечными моделями формул свободной теории и автоматами над деревьями [87]. На основе этого соответствия можно создать следующую процедуру построения автоматов над деревьями по конечным моделям. По конечной модели $\mathcal{M}$ для каждого предикатного символа $P \in \Sigma_P$ строится автомат $A_P = \langle |\mathcal{M}|, \Sigma_F, \mathcal{M}(P), \Delta \rangle$; для всех автоматов определено общее отношение переходов Δ — для каждого $f \in \Sigma_F$ с арностью $\sigma_1 \times \ldots \times \sigma_n \mapsto \sigma$ и для каждого $x_i \in |\mathcal{M}|_{\sigma_i}$ положим $\Delta\big(f(x_1, \ldots, x_n)\big) = \mathcal{M}(f)(x_1, \ldots, x_n)$.

Теорема 2. Для любого построенного автомата A_P справедливо следующее утверждение:

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

Практическая ценность этого результата заключается в том, что построение автомата над деревьями по формуле эквивалентно поиску конечной модели для неё. Таким образом, ряд инструментов, таких как Mace4 [88], Kodkod [89], Paradox [90], а также cvc5 [91] и Vampire [92] могут быть использованы для поиска конечных моделей формул свободной теории и, как следствие, для автоматического построения автоматов над деревьями.

Большинство из этих инструментов реализуют кодирование в SAT: конечный домен и функции над ним кодируются в битовое представление, после этого по нему получают формулу логики высказываний, которую передают в SAT-решатель. Данные инструменты применяются для задач верификации [93], а также для построения бесконечных моделей формул первого порядка [94].

1.6 Выводы

Ключевую роль в формальных методах, в особенности, в статическом анализе, играет задача автоматического вывода индуктивных инвариантов программ. Не смотря на то, что существует некоторое количество весьма проработанных методов вывода индуктивных инвариантов, и каждый год по этой теме появляются публикации на различных конференциях по информатике и языкам программирования ранга A* (POPL, PLDI и CAV и др.). Также каждый год проводятся соревнования между соответствующими инструментами, всё ещё остаётся открытой проблема следующая проблема: как лучше выражать индуктивные инварианты программ. Проблема подбора наилучшего представления инвариантов состоит в том, чтобы, с одной стороны, были выразимы инварианты реальных программ, а, с другой стороны, существовала бы эффективная процедура вывода инвариантов. Эта проблема стоит наиболее остро в контексте алгебраических типов данных, для которых классические способы представлять инварианты крайне малоэффективны, а если инвариант не представим, то алгоритм его вывода в этом представлении не будет завершаться. Это делает данное диссертационное исследование востребованным и актуальным.

Глава 2. Вывод регулярных инвариантов

Основным вкладом данной главы является новый метод автоматического вывода индуктивных инвариантов систем над АТД при помощи инструментов автоматического доказательства теорем. В разделе 2.1 представлен метод и доказана его корректность для систем дизъюнктов упрощённого вида (без ограничений), а в разделе 2.2 — для произвольных систем. В разделе 2.3 рассмотрен класс регулярных инвариантов, которые могут быть выводимы при помощи предложенного метода. В разделе 2.4 описано, как предложенный метод может быть применён для вывода регулярных инвариантов при помощи инструментов поиска конечных моделей. В отличие от классических элементарных инвариантов, регулярные инварианты, основанные на автоматах над деревьями, позволяют выражать рекурсивные отношения, в частности, свойства алгебраических термов произвольной глубины. Как указано в разделе 2.2, предложенный метод также может быть совмещён с общими инструментами автоматического доказательства теорем.

2.1 Метод для систем без ограничений в дизъюнктах

Основная идея метода заключается в следующем. Если у системы дизъюнктов Хорна над АТД без ограничений есть модель в свободной теории, то она безопасна и этой модели соответствует некоторый индуктивный инвариант.

Пример 2. Рассмотрим следующую систему дизъюнктов Хорна над алгебраическим типом чисел Пеано $Nat ::= Z \mid S\,Nat$. Эта система кодирует предикат чётности чисел Пеано $even$ и свойство: «никакие два следующих друг за другом натуральных числа не могут быть чётными одновременно».

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

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

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

Хотя эта простая система безопасна, у неё нет классического символьного инварианта, что будет показано в главе 5.

Эта система может быть переписана в следующую эквивалентную систему без ограничений в дизъюнктах.

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

Ей соответствует следующая формула в свободной теории.

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

Эта формула выполняется следующей конечной моделью $\mathcal{M}$:

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

Лемма 1 (Корректность). Пусть система дизъюнктов Хорна $\mathcal{P}$ с неинтерпретированными предикатами $\mathcal{R} = \{P_1, \ldots, P_k\}$ без ограничений в дизъюнктах выполняется в некоторой модели $\mathcal{M}$, т. е. $\mathcal{M} \models C$ для всех $C \in \mathcal{P}$. Пусть также справедливо следующее:

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

Тогда $\langle \mathcal{H}, X_1, \ldots, X_k \rangle$ является индуктивным инвариантом $\mathcal{P}$.

Доказательство. Все дизъюнкты имеют вид:

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

Возьмём некоторый подходящий по сортам кортеж замкнутых термов $\overline{x}$. Тогда из $\mathcal{M} \models \forall C$, по определению X_i, следует, что

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

где H' — соответствующая подстановка для H. По определению выполнимости дизъюнкта Хорна из этого следует, что

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

□

Таким образом, для примера выше мы строим из конечной модели множество $X \triangleq \{t \mid \mathcal{M}[\![t]\!] = 0\} = \{S^{2n}(Z) \mid n \geqslant 0\}$, которое является безопасным индуктивным инвариантом исходной системы.

2.2 Метод для систем с ограничениями в дизъюнктах

Для системы с ограничениями в дизъюнктах можно построить эквивыполнимую систему без ограничений в дизъюнктах следующим образом. Без ограничения общности предположим, что ограничение каждого дизъюнкта содержит отрицания только над атомами. Литералы равенств термов могут быть устранены при помощи унификации [95]. Каждый литерал неравенства вида $\neg(t =_\sigma u)$ заменим на атомарную формулу $diseq_\sigma(t, u)$. Для каждого алгебраического типа (C, σ) введём новый неинтерпретированный символ $diseq_\sigma$ и добавим его в множество реляционных символов $\mathcal{R}' \triangleq \mathcal{R} \cup \{diseq_\sigma \mid \sigma \in \Sigma_S\}$.

Далее по системе $\mathcal{P}$ построим систему дизъюнктов $\mathcal{P}'$ над $\mathcal{R}'$ следующим образом. Для каждого алгебраического типа (C, σ) в $\mathcal{P}'$ добавим следующие дизъюнкты для $diseq_\sigma$:

$\top \to diseq_\sigma(c(\overline{x}), c'(\overline{x}'))$ для всех различных конструкторов c и $c' \in C$ сорта σ

и

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

для всех конструкторов c сорта σ, всех i и всех x, y сорта σ'.

Для каждого сорта $\sigma \in \Sigma_S$ введём диагональное множество $\mathcal{D}_\sigma \triangleq \{(x,y) \in |\mathcal{H}|^2_\sigma \mid x \neq y\}$.

Хорошо известно, что универсально замкнутые дизъюнкты Хорна имеют наименьшую модель, которая является денотационной семантикой программы, моделируемой системой дизъюнктов [21]. Эта наименьшая модель является наименьшей неподвижной точка оператора перехода. Из этого тривиально следует следующая лемма.

Лемма 2. Наименьший индуктивный инвариант дизъюнктов для $diseq_\sigma$ является кортежем отношений $\mathcal{D}_\sigma$.

Простым следствием этой леммы является следующий факт.

Лемма 3. Для системы дизъюнктов Хорна $\mathcal{P}'$, полученной при помощи описанной выше трансформации, если $\langle \mathcal{H}, X_1, \ldots, X_k, Y_1, \ldots, Y_n \rangle \models \mathcal{P}'$, то $\langle \mathcal{H}, X_1, \ldots, X_k, \mathcal{D}_{\sigma_1}, \ldots, \mathcal{D}_{\sigma_n} \rangle \models \mathcal{P}'$ (отношения Y_i и $\mathcal{D}_{\sigma_i}$ интерпретируют предикатные символы $diseq_{\sigma_i}$).

Пример 3. Система дизъюнктов $\mathcal{P} = \{Z \neq S(Z) \to \bot\}$ трансформируется в следующую систему $\mathcal{P}'$:

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

Корректность трансформации, приведённой в данном разделе, доказывается следующей теоремой.

Теорема 3 (Корректность). Пусть $\mathcal{P}$ — система дизъюнктов Хорна, а $\mathcal{P}'$ — система дизъюнктов, полученная описанной выше трансформацией. Если существует модель системы $\mathcal{P}'$ в свободной теории, то у исходной системы $\mathcal{P}$ есть индуктивный инвариант.

Доказательство. Без ограничения общности можно предположить, что каждый дизъюнкт $C \in \mathcal{P}$ имеет следующий вид:

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

В $\mathcal{P}'$ этот дизъюнкт трансформируется в следующий дизъюнкт:

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

Таким образом, каждое предложение в $\mathcal{P}'$ не содержит ограничений (т. к. правила $diseq$ также не содержат ограничений), а следовательно по лемме 1 у $\mathcal{P}'$ есть некоторый индуктивный инвариант $\langle \mathcal{H}, X_1, \ldots, X_k, U_1, \ldots, U_n \rangle$. Тогда по лемме 3 имеем $\langle \mathcal{H}, X_1, \ldots, X_k, \mathcal{D}_{\sigma_1}, \ldots, \mathcal{D}_{\sigma_n} \rangle \models C'$ для каждого $C' \in \mathcal{P}'$. Однако очевидно следующее:

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

Это означает, что $\langle \mathcal{H}, X_1, \ldots, X_k \rangle \models C$ для каждого $C \in \mathcal{P}$ — желаемый индуктивный инвариант исходной системы. □

Использование метода для вывода инвариантов. Для проверки выполнимости формул первого порядка могут быть использованы инструменты автоматического доказательства теорем, строящие насыщения, такие как Vampire [96], E [97] и Zipperposition [98]. Однако насыщения не дают эффективный класс инвариантов, поскольку даже проверка принадлежности кортежа замкнутых термов множеству, выраженному насыщением, неразрешима [99]. По этой причине насыщения в качестве основы для самостоятельного класса инвариантов не рассматриваются в данной работе. Однако изучение их подклассов, как и создание процедур автоматического вывода для инвариантов в них являются многообещающими.

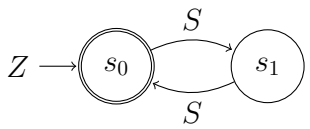
О применении предложенного метода для вывода инвариантов в более узком классе регулярных инвариантов повествуют следующие разделы.

2.3 Регулярные инварианты

Определение 9 (Reg). Будем говорить, что n-автомат A над Σ_F *выражает отношение* $X \subseteq |\mathcal{H}|_{\sigma_1} \times \ldots \times |\mathcal{H}|_{\sigma_n}$, если $X = \mathcal{L}(A)$. Отношение X, для которого существуют выражающий его автомат над деревьями, называется *регулярным отношением*. Класс регулярных отношений будет обозначаться Reg.

Пусть $\mathcal{P}$ — система дизъюнктов Хорна. Если $\overline{X} = \langle X_1, \ldots, X_n \rangle$, где каждый X_i регулярен, и $\langle \mathcal{H}, \overline{X} \rangle \models C$ для всех $C \in \mathcal{P}$, тогда $\langle \mathcal{H}, \overline{X} \rangle$ называется *регулярным инвариантом* $\mathcal{P}$.

Пример 4. Система дизъюнктов Хорна из примера 2 имеет регулярный инвариант $\langle \mathcal{H}, \mathcal{L}(A) \rangle$, где A — это 1-автомат $\langle \{s_0, s_1, s_2\}, \Sigma_F, \{s_0\}, \Delta \rangle$ со следующим отношением перехода Δ:



Множество $\mathcal{L}(A) = \{Z, S(S(Z)), S(S(S(S(Z)))), \ldots\} = \{S^{2n}(Z) \mid n \geqslant 0\}$ очевидным образом удовлетворяет всем дизъюнктам системы.

Пример 5. Рассмотрим следующую систему дизъюнктов, у которой есть несколько индуктивных инвариантов.

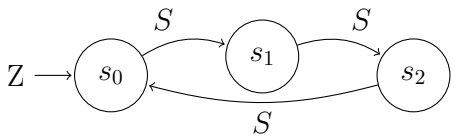
$$\begin{aligned} x = Z \wedge y = S(Z) &\to inc(x, y) \\ x = S(x') \wedge y = S(y') \wedge inc(x', y') &\to inc(x, y) \\ x = S(Z) \wedge y = Z &\to dec(x, y) \\ x = S(x') \wedge y = S(y') \wedge dec(x', y') &\to dec(x, y) \\ inc(x, y) \wedge dec(x, y) &\to \bot \end{aligned}$$

Эта система имеет следующий очевидный элементарный инвариант:

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

Данный инвариант является наиболее сильным из всех возможных, т. к. выражает денотационную семантику inc и dec соответственно. Эти отношения нерегулярны, то есть не существует 2-автоматов, представляющих эти отношения [35].

Однако эта система дизъюнктов имеет также другой, менее очевидный, регулярный инвариант, порождённый двумя 2-автоматами $\left\langle \{s_0, s_1, s_2, s_3\}, \Sigma_F, S_*, \Delta \right\rangle$ с конечными состояниями соответственно $S_{inc} = \{\langle s_0, s_1 \rangle, \langle s_1, s_2 \rangle, \langle s_2, s_0 \rangle\}$, $S_{dec} = \{\langle s_1, s_0 \rangle, \langle s_2, s_1 \rangle, \langle s_0, s_2 \rangle\}$ и с правилами перехода, имеющими следующий вид:



Автомат для inc проверяет, что $(x \, mod \, 3, \, y \, mod \, 3) \in \{(0,1), (1,2), (2,0)\}$, а автомат для dec проверяет, что $(x \, mod \, 3, \, y \, mod \, 3) \in \{(1,0), (2,1), (0,2)\}$. Эти отношения аппроксимируют сверху денотационную семантику inc и dec и при этом доказывают невыполнимость формулы $inc(x, y) \wedge dec(x, y)$.

Таким образом, хотя многие отношения нерегулярны, у программ могут существовать неочевидные регулярные инварианты.

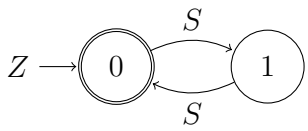
Более подробно свойства регулярных инвариантов рассмотрены в главе 5.

Рисунок 2.1 — Метод вывода регулярного инварианта для системы дизъюнктов Хорна над АТД.

2.4 Специализация метода для вывода регулярных инвариантов

Предложенный в прошлых разделах метод может быть специализирован для вывода регулярных инвариантов, как показано на рисунке 2.1. При помощи трансформаций из разделов 2.1 и 2.2 по системе дизъюнктов Хорна над АТД можно получить эквивыполнимую формулу первого порядка над свободной теорией. Если запустить для неё инструмент поиска конечных моделей, то при помощи классического построения — см. теорему 2 об изоморфизме между конечными моделями и автоматами над деревьями — можно получить автомат над деревьями, выражающий регулярный инвариант исходной системы дизъюнктов Хорна. Корректность всего подхода гарантируется теоремами 3 и 2.

Например, по конечной модели из раздела 2.1 для примера *Even* будет получен следующий автомат A_{Even}, изоморфный представленному в примере 4.



На практике это означает, что индуктивные инварианты систем дизъюнктов Хорна над АТД можно строить при помощи *инструментов поиска конечных моделей*, таких как Mace4 [88], Kodkod [89], Paradox [90], а также cvc5 [91] и Vampire [92].

2.5 Выводы

Предложенный метод позволяет свести задачу поиска индуктивного инварианта системы дизъюнктов Хорна с АТД к задаче проверки выполнимости

формулы универсального фрагмента логики первого порядка. Поэтому совместно с предложенным методом могут быть использованы произвольные инструменты автоматического доказательства теорем, такие как Vampire [96], E [97] и Zipperposition [98]. Такие инструменты возвращают доказательства выполнимости в виде насыщений, которые позволяют выражать широкий класс инвариантов. Однако проверка того, что насыщение выражает индуктивный инвариант заданной системы, неразрешима, поэтому использование насыщений для выражения индуктивных инвариантов не представляется возможным. Также совместно с предложенным методом могут быть использованы инструменты поиска конечных моделей, такие как, например, Mace4 [88], Kodkod [89], Paradox [90], а также cvc5 [91] и Vampire [92] в соответствующих режимах. Композиция предложенного метода и инструмента поиска конечных моделей может выводить регулярные инварианты, которые позволяют выражать рекурсивные отношения и представлять инварианты некоторых систем, для которых классических символьных инвариантов не существует. Кроме того, проверка того, что заданный автомат над деревьями выражает регулярный инвариант заданной системы, разрешима. Ограничением регулярных инвариантов является то, что они не позволяют представлять синхронные отношения, такие как инкремент чисел Пеано, поэтому существуют системы, у которых есть классический символьный инвариант, но нет регулярных. Более богатый класс *синхронных* регулярных инвариантов, решающий эту проблему, а также новый метод вывода инвариантов для этого класса, рассмотрены в следующей главе.

Глава 3. Вывод синхронных регулярных инвариантов

В качестве расширения автоматов над деревьями, способного выражать синхронные отношения, часто применяют синхронные автоматы над деревьями. Выразительная сила класса синхронных автоматов зависит от схемы свёртки термов, на которой этот класс построен. В разделе 3.1 рассмотрен класс синхронных регулярных инвариантов, основанный на синхронных автоматах, построенных по произвольной схеме свёртки. В частности, рассмотрены синхронные регулярные инварианты, основанные на полной свёртке, которые позволяют выражать большой класс синхронных отношений. В разделе 3.2 предложен метод вывода синхронных регулярных инвариантов, основанный на трансформации системы дизъюнктов в декларативное описание автомата, задающего инвариант.

3.1 Синхронные регулярные инварианты

Синхронные автоматы над деревьями со стандартной [35] и полной [28] свёрткой часто рассматриваются как естественное расширение классических автоматов над деревьями для выражения синхронных отношений, таких как равенство и неравенство термов. В данном разделе определены автоматы над деревьями с произвольной свёрткой и доказаны их базовые свойства.

3.1.1 Синхронные автоматы над деревьями

Определение 10. *Свёртка (convolution) термов* — это вычислимая биективная функция из $\mathcal{T}(\Sigma_F)^{\leqslant k}$ в $\mathcal{T}(\Sigma_F^{\leqslant k})$ для некоторого $k \geqslant 1$.

Определение 11 (см. [28; 35]). *Стандартная свёртка (standard convolution)* σ_{sc} *термов* определяется следующим образом:

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

Пример 6. Рассмотрим следующее применение стандартной свёртки к кортежу термов:

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

Определение 12 (см. [28]). *Полная свёртка (full convolution)* σ_{fc} *термов* определяется следующим образом:

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

Определение 13. Множество кортежей термов X называется σ-*свёрточным регулярным языком*, если существует автомат над деревьями A такой, что $\mathcal{L}(A) = \{\sigma(\overline{t}) \mid \overline{t} \in X\} \triangleq \sigma(X)$.

Классом языков REG_σ называется множество всех σ-свёрточных регулярных языков. Обозначим посредством REG_+ класс $\mathrm{REG}_{\sigma_{sc}}$ и $\mathrm{REG}_\times$ — класс $\mathrm{REG}_{\sigma_{fc}}$.

Лемма 4. Пусть L — язык кортежей арности 1. Тогда справедливо, что $L \in \mathrm{REG}_\times \Leftrightarrow L \in \mathrm{REG}$.

Доказательство. По определению имеем, что $\sigma_{fc}\big(f(\overline{a})\big) \triangleq \langle f\rangle\,\big(\sigma_{fc}(\overline{b}) \mid b \in (\overline{a})\big)$. Другими словами, $\sigma_{fc}\big(f(a_1,\ldots,a_n)\big) \triangleq f\big(\sigma_{fc}(a_1),\ldots,\sigma_{fc}(a_n)\big)$. Следовательно, $\sigma_{fc}(t) = t$ для всех термов t, а значит $\sigma_{fc}(L) = L$ и $L \in \mathrm{REG}_\times$, $L = \sigma_{fc}(L) \in \mathrm{REG}$. □

Пример 7. Рассмотрим сигнатуру Σ_F бинарных деревьев, имеющую два конструктора — $Node$ и $Leaf$ арности 2 и 0 соответственно. Рассмотрим автомат $A = \big\langle\{\top,\bot\}, \Sigma_F^{\leqslant 2}, \{\bot\}, \Delta\big\rangle$ с отношением перехода Δ:

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

где φ и ψ проходят по множеству всех состояний. Этот автомат позволяет выразить отношение неравенства при помощи стандартной свёртки. Иными словами, $\mathcal{L}(A) = \{\sigma_{sc}(x,y) \mid x,y \in \mathcal{T}(\Sigma_F), x \neq y\}$.

Пример 8 (*lt*). Рассмотрим сигнатуру Σ_F натуральных чисел Пеано, имеющую два конструктора Z и S арности 0 и 1 соответственно, и следующее множество, задающее порядок на этих числах:

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

Возьмём автомат $A = \langle \{\bot, \top\}, \Sigma_F^{\leqslant 2}, \{\top\}, \Delta \rangle$ с отношением перехода Δ:

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

где $\varphi \in \{\top, \bot\}$ проходит по множеству всех состояний. Этот автомат позволяет выразить отношение порядка при помощи стандартной свёртки. Иными словами, $\mathcal{L}(A) = \{\sigma_{sc}(S^n(Z), S^m(Z)) \mid n < m\}$.

3.1.2 Замкнутость относительно булевых операций

Регулярные языки со свёрткой замкнуты относительно всех булевых операций вне зависимости от свёртки. В сущности, доказательства и соответствующие конструкции для классических автоматов подходят и для автоматов со свёрткой. В данном разделе будем обозначать с помощью k размерность кортежей языков из REG_σ.

Теорема 4. Класс языков REG_σ с произвольной свёрткой σ замкнут относительно дополнения.

Доказательство. Пусть язык $L \in \mathrm{REG}_\sigma$. Тогда без ограничения общности можно утверждать, что существует детерминированный автомат $A = \langle S, \Sigma_F^{\leqslant k}, S_F, \Delta \rangle$ такой, что $\mathcal{L}(A) = \sigma(L)$. Рассмотрим автомат для языка дополнения $A^c = \langle S, \Sigma_F^{\leqslant k}, S \setminus S_F, \Delta \rangle$. Верно, что $\mathcal{L}(A^c) = \overline{\mathcal{L}(A)} = \overline{\sigma(L)} = \sigma(\overline{L})$ (последнее следует из того, что σ — биективная функция). Таким образом имеем, что $\overline{L} \in \mathrm{REG}_\sigma$. □

Теорема 5. Класс языков REG_σ с произвольной свёрткой σ замкнут относительно пересечения.

Доказательство. Рассмотрим $L_1, L_2 \in \mathrm{REG}_\times$. Тогда имеются детерминированные автоматы $A = \langle S^A, \Sigma_F^{\leqslant k}, S_F^A, \Delta^A \rangle$ и $B = \langle S^B, \Sigma_F^{\leqslant k}, S_F^B, \Delta^B \rangle$ такие, что $\mathcal{L}(A) = L_1$ и $\mathcal{L}(A) = L_2$. Пересечение языков $L_1 \cap L_2$ распознаётся автоматом

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

где отношение перехода Δ определяется так:

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

Из биективности σ следует, что $\mathcal{L}(C) = \mathcal{L}(A) \cap \mathcal{L}(B) = \sigma(L_1) \cap \sigma(L_2) = \sigma(L_1 \cap L_2)$, следовательно $L_1 \cap L_2 \in \text{REG}_\sigma$. □

Теорема 6. Класс языков REG_σ с произвольной свёрткой σ замкнут относительно объединения.

Доказательство. Данное утверждение напрямую следует из теорем 4 и 5 и закона де Моргана, применённого к множествам L_1 и L_2: $L_1 \cup L_2 = (L_1^c \cap L_2^c)^c$. □

3.1.3 Разрешимость проблем пустоты и принадлежности терма

Перенесём разрешающие процедуры для классических автоматов над деревьями на автоматы со свёрткой.

Теорема 7. Пусть σ — произвольная свёртка и $X \in \text{REG}_\sigma$. Тогда задача проверки пустоты множества X является разрешимой.

Доказательство. Пусть A — автомат над деревьями такой, что $\mathcal{L}(A) = \sigma(X)$, $X = \varnothing \Leftrightarrow \mathcal{L}(A) = \varnothing$. Пустоту языка классического автомата над деревьями позволяет проверить процедура из [35, теор. 1.7.4], которая работает за линейное время от размера автомата. □

Теорема 8. Пусть σ — произвольная свёртка и $X \in \text{REG}_\sigma$. Задача принадлежности кортежа замкнутых термов множеству X является разрешимой.

Доказательство. Возьмём кортеж замкнутых термов $\bar{t}$ и A — автомат над деревьями такой, что $\mathcal{L}(A) = \sigma(X)$. Тогда верно следующее:

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

Следовательно, искомая процедура заключается в вычислении σ на кортеже $\bar{t}$ и проверке принадлежности результата языку автомата A при помощи процедуры из [35, теор. 1.7.2]. □

3.2 Вывод инвариантов путём декларативного описания задающего инвариант автомата

В данном разделе предложена процедура Δ, которая по системе дизъюнктов Хорна строит формулу первого порядка, декларативно описывающую

индуктивный инвариант исходной системы. Формула $\Delta(\mathcal{P})$ имеет конечную модель тогда и только тогда, когда оригинальная система $\mathcal{P}$ имеет индуктивный инвариант в классе $\text{REG}_\times$. Из этого легко следует метод вывода синхронных регулярных инвариантов, заключающийся в применении произвольной процедуры поиска конечных моделей к результату применения процедуры Δ к системе дизъюнктов Хорна. Для определения процедуры Δ вводится языковая семантика формул, позволяющая говорить о семантике языков формул, построенных из языков предикатов.

3.2.1 Языковая семантика формул

Формула $\Phi = \forall x_1 \ldots \forall x_n . \varphi(x_1, \ldots, x_n)$ находится в *сколемовской нормальной форме (СНФ)*, если φ является бесквантворной формулой со свободными переменными $x_1, \ldots, x_n$. Известно, что любая формула в языке первого порядка может быть приведена к сколемовской нормальной форме с помощью процедуры сколемизации, поэтому достаточно определить языковую семантику для формул в СНФ.

Определение 14. Кортеж термов $\langle t_1, \ldots, t_k \rangle$ называется (n,k)-*шаблоном*, если каждый его элемент t_i зависит не более чем от n переменных из общего набора переменных данного кортежа.

Определение 15. Будем называть шаблон *линейным*, если каждая переменная входит не более чем в один терм кортежа, и в любой терм входит не более одного раза. В противном случае будем называть шаблон *нелинейным*.

Определение 16. *Подстановкой замкнутых термов* $u = \langle u_1, \ldots, u_n \rangle$ в (n,k)-шаблон $t = \langle t_1, \ldots, t_k \rangle$ назовем кортеж замкнутых термов, полученный подстановкой термов u_i на место переменных x_i для $i = 1, \ldots, n$:

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

Определение 17. *Нижний остаток языка* L по (n,k)-шаблону t — это n-арный язык $L/t \triangleq \{u \in \mathcal{T}(\Sigma_F)^n \mid t[u] \in L\}$.

Пример 9. Рассмотрим сигнатуру чисел Пеано, содержащую два функциональных символа Z и S, имеющих арность 0 и 1 соответственно.

Кортеж $\langle x_1, S(x_2), Z \rangle$ является линейным (2,3)-шаблоном.

Кортеж $\langle S(x_1), x_1\rangle$ является нелинейным (1,2)-шаблоном.

Подстановка кортежа термов $u = \langle Z, S(Z)\rangle$ в $(2,3)$-шаблон $t = \langle S(x_1), S(S(x_2), Z)\rangle$ является кортежем $t[u] = \langle S(Z), S(S(S(Z))), Z\rangle$.

Определение 18. Пусть каждому неинтерпретированному предикатному символу p соответствует некоторый язык кортежей термов, обозначаемый $L[\![p]\!]$. Язык равенства определён как $L[\![=]\!] = \{(x,x) \mid x \in \mathcal{T}(\Sigma_F)\}$. *Языковой семантикой* формулы в СНФ $\forall x_1 \ldots \forall x_n.\varphi(x_1, \ldots, x_n)$ назовём язык $L[\![\varphi]\!]$, определённый индуктивно следующим образом:

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

Определение 19. Формула в СНФ $\Phi = \forall x_1 \ldots \forall x_n.\varphi(x_1, \ldots, x_n)$ *выполнима в языковой семантике* ($L \models \Phi$), если $L[\![\neg\varphi]\!] = \varnothing$.

Теорема 9. Формула в СНФ выполнима в языковой семантике тогда и только тогда, когда она выполнима в семантике Тарского.

Доказательство. Возьмём $\Phi = \forall x_1 \ldots \forall x_n.\varphi(x_1, \ldots, x_n)$. По теореме Эрбрана формула Φ выполнима в семантике Тарского тогда и только тогда, когда у нее существует эрбрановская модель $\mathcal{H}$. Возьмём $L[\![p]\!] = \mathcal{H}[\![p]\!]$ и построим доказательство индукцией по структуре формулы:

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

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

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

Для доказательства индукционного перехода для дизъюнкции можно воспользоваться законом Де Моргана. □

Теорема 10. Пусть $L \in \text{REG}_\times$ — язык кортежей размера n. Тогда нижний остаток L/t относительно линейного шаблона $t = \langle x_1, \ldots, x_{i-1}, f(y_1, \ldots, y_m), x_{i+1}, \ldots x_n \rangle$ также принадлежит классу $\text{REG}_\times$.

Доказательство. Не ограничивая общности, рассмотрим шаблон $t = \langle f(y_1, \ldots, y_m), x_2, \ldots x_n \rangle$. Пусть $\sigma_{fc}(L) = \mathcal{L}(A)$, где $A = \langle S, \Sigma_F^{\leqslant n}, S_F, \Delta \rangle$. Рассмотрим автомат $A' = \langle S', \Sigma_F^{\leqslant n-1+m}, S'_F, \Delta' \rangle$, каждое состояние которого хранит не более $n-1$ функциональных символов и не более m^n состояний автомата A, то есть $S' = \Sigma^{\leqslant n-1} \times S^{\leqslant m^n}$.

Далее, определим автомат A' таким образом, чтобы выполнялось следующее свойство:

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

Определим конечные состояния автомата A' так:

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

Тогда по свойству 3.1 имеем следующее:

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

Чтобы определить отношение перехода Δ', рассмотрим раскрутку вычисления A':

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

где

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

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

Для того, чтобы выполнялось свойство 3.1, левая часть равенства 3.2 должна быть равна следующей паре:

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

Второй элемент данной пары по определению равен следующему выражению:

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

Каждый элемент $A\left[\sigma_{fc}\left(\overline{b}\right)\right]$ в последнем выражении гарантированно присутствует среди аргументов отношения перехода Δ' (обозначенных $\overline{a}'$), поэтому из приведённых равенств можно построить корректное определение Δ', заменив все вхождения $A\left[\sigma_{fc}\left(\overline{b}\right)\right]$ в последнем выражении и $\overline{a}'$ на свободные переменные состояний.

Для автомата A' верно по построению, что $\mathcal{L}(A) = \sigma_{fc}(L/t)$. □

Теорема 11. Пусть $L \in \text{REG}_\times$, а кортеж t является (k,n)-шаблоном. Тогда выполняется $L/t \in \text{REG}_\times$.

Доказательство. Язык L/t можно линеаризовать, то есть представить в виде пересечения нижних остатков языка L по линейным шаблонам и языков для равенств некоторых переменных. Заключение теоремы следует из теоремы 10 о замкнутости $\text{REG}_\times$ относительно нижних остатков по линейным шаблонам и теоремы 5 о замкнутости этого класса относительно пересечений. □

3.2.2 Алгоритм построения декларативного описания инварианта

В данном разделе приведено описание алгоритма Δ-трансформации системы дизъюнктов Хорна над АТД в формулу логики первого порядка над свободной теорией, по конечной модели которой можно построить синхронный регулярный инвариант исходной системы дизъюнктов.

Искомый алгоритм начинает с устранения ограничений из дизъюнктов при помощи алгоритма, представленного в разделе 2.2.

Далее, по системе дизъюнктов Хорна $\mathcal{P}$ с предикатами $\mathcal{R}$ алгоритм Δ строит формулу языка первого порядка в сигнатуре $\Sigma' = \langle \Sigma'_S, \Sigma'_F, \Sigma'_P \rangle$, где

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

Здесь введены сорт S для состояний автоматов и сорт $\mathcal{F}$ для конструкторов АТД. Функциональные символы $delta$ введены для отношений переходов автоматов, а предикатные символы $Reach$ и $Final$ — для достижимых и конечных состояний, соответственно. Для каждого предиката, атома и дизъюнкта строятся соответствующие автоматы. Функциональный символ $prod_n$ арности $S^n \mapsto S$ позволяет строить состояния, являющиеся кортежами других состояний. Функциональный символ $delay_{n,m}$ арности $\mathcal{F}^n \times S^m \mapsto S$ позволяет строить состояния, являющиеся кортежами конструкторов и состояний. Алгоритм Δ возвращает конъюнкцию из декларативных описаний синхронных автоматов для каждого дизъюнкта и для каждого атома, которые определены далее.

Пусть дан дизъюнкт C. По определению выполнимости в языковой семантике имеем $L \models C \Leftrightarrow L[\![\neg C]\!] = \varnothing$. Таким образом, декларативным описанием для дизъюнкта будет формула первого порядка, выражающая $L[\![\neg C]\!] = \varnothing$. Пусть $\neg C \Leftrightarrow A_1 \wedge \ldots \wedge A_{n-1} \wedge \neg A_n$, где A_i — атомарные формулы. Пусть для каждого A_i имеется декларативное описание соответствующего атому автомата с символами $\langle delta_{A_i}, Final_{A_i} \rangle$. Определим автомат для дизъюнкта C с символами $\langle delta_C, Final_C \rangle$ при помощи конструкции из доказательства теоремы 5 о замкнутости автоматов относительно пересечения. Декларативное описание для дизъюнкта является конъюнкцией универсальных замыканий по всем свободным переменным следующих четырёх формул:

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

Здесь все x имеют сорт $\mathcal{F}$, а все q — сорт S. Верхние индексы j переменных состояний q_i^j соответствуют порядковому номеру автомата атома A_j, в

котором используется переменная состояния. Первые две формулы кодируют конструкцию произведения автоматов из теоремы 5. Третья формула определяет множество состояний, достижимых автоматом для дизъюнкта. Последняя формула кодирует пустоту языка дизъюнкта («нет ни одного одновременно конечного и достижимого состояния»).

Декларативное описание автомата для атома описано в доказательстве теорем 10 и 11. Кортежи вида $\langle\langle f_2, \ldots, f_n\rangle, \overline{q}\rangle$, где f имеет сорт $\mathcal{F}$, а q — сорт S, кодируются при помощи функциональных символов $delay$.

3.2.3 Корректность и полнота

Теорема 12. У системы $\Delta(\mathcal{P})$ конечная модель существует тогда и только тогда, когда у системы дизъюнктов Хорна $\mathcal{P}$ существует решение, представленное полносверточными синхронными автоматами над деревьями.

Доказательство. Доказательство следует из построения $\Delta(\mathcal{P})$, теорем о замкнутости относительно булевых операций и нижнего остатка 4, 5, 11, теоремы 9 о выполнимости СНФ в языковой семантике и того факта, что всякая система дизъюнктов Хорна сводится в СНФ переименованием переменных. □

3.2.4 Пример

Рассмотрим описанную в данной главе трансформацию на примере следующей системы дизъюнктов Хорна с неинтерпретированным предикатным символом lt, задающим отношение строгого порядка на числах Пеано:

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

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

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

Дизъюнкт $C1$ эквивалентен атомарной формуле $A_1 = lt(Z, S(x))$. Для автомата атомарной формулы A_1 в $\Delta(\mathcal{P})$ окажутся универсальные замыкания следующих формул, сконструированные на основе автомата для предикатного символа lt:

$$delta_{A_1}(Z) = delta_{lt}(Z)$$

$$delta_{A_1}(S, q) = delta_{lt}(S, q)$$

$$Final_{A_1}(q) \leftrightarrow Final_{lt}(delta_{lt}(Z, S, q))$$

Для дизъюнкта $C1$ в $\Delta(\mathcal{P})$ окажутся следующие формулы автомата $(delta_{C1}, Final_{C1})$, сконструированного на основе автомата для атомарной формулы A_1:

$$delta_{C1}(f,q) = delta_{A_1}(f,q)$$
$$Final_{C1}(q) \leftrightarrow \neg Final_{A_1}(q)$$

Также в $\Delta(\mathcal{P})$ будут входить следующие условия пустоты языка автомата дизъюнкта $C1$, которые служат гарантом его выполнения:

$$Reach_{C1}(q) \rightarrow Reach_{C1}(delta_{C1}(f,q))$$
$$Reach_{C1}(q) \wedge Final_{C1}(q)) \rightarrow \bot$$

Дизъюнкт $C2$ состоит из двух атомарных формул: $A_2 = lt(x,y)$ и $A_3 = lt(S(x),S(y))$. Автомат для атомарной формулы A_2 совпадает с автоматом предикатного символа lt, для атомарной формулы A_3 добавим в $\Delta(\mathcal{P})$ автомат $(delta_{A_3}, Final_{A_3})$, сконструированный на основе автомата для предикатного символа lt:

$$delta_{A_3}(f,g,q) = delta_{lt}$$
$$Final_{A_3}(q) \leftrightarrow Final_{lt}(delta_{lt}(S,S,q))$$

Для дизъюнкта $C2$ добавим в $\Delta(\mathcal{P})$ автомат $(delta_{C2}, Final_{C2})$, сконструированный на основе автомата для предикатного символа lt и автомата для атомарной формулы A_3:

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

Добавим в $\Delta(\mathcal{P})$ условия пустоты языка автомата $(delta_{C2}, Final_{C2})$ для гарантии выполнения дизъюнкта $C2$:

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

Дизъюнкт $C3$ состоит из двух атомарных формул $A4 = lt(x,y)$ и $A5 = lt(y,x)$. Автомат для атомарной формулы $A4$ полностью совпадает с автоматом для предикатного символа lt. Автомат для атомарной формулы $A5$ отличается

от автомата для предикатного символа lt только порядком аргументов, и после взятия остатка по такому линейному паттерну получается автомат, идентичный lt.

Для дизъюнкта $C3$ добавим в $\Delta(\mathcal{P})$ автомат $(delta_{C3}, Final_{C3})$, сконструированный на основе автомата для предикатного символа lt:

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

Добавим в $\Delta(\mathcal{P})$ условия пустоты языка автомата $(delta_{C3}, Final_{C3})$ для гарантии выполнения дизъюнкта $C3$:

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

Запустив на формуле $\Delta(\mathcal{P})$ инструмент поиска конечных моделей, можно извлечь из интерпретаций $delta_{lt}$ и $Final_{lt}$ синхронный регулярный инвариант исходной системы дизъюнктов Хорна, основанный на автомате $A_{lt} = \langle\{0,1,2\}, \Sigma_F, \{1\}, \Delta\rangle$, где для $q \in \{1,2\}$:

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

Языком этого автомата является множество пар чисел Пеано, где первое число строго меньше второго.

3.3 Выводы

Рассмотренный класс синхронных регулярных инвариантов с полной свёрткой включает в себя регулярные инварианты, а также большой класс классических символьных инвариантов. Поскольку используется *полная* свёртка, любые операции с такими автоматами будут приводить к экспоненциальному «взрыву» сложности. Следует отметить, что хотя предложенный метод вывода таких инвариантов теоретически существует, его эффективность необходимо проверить на практике, что выполнено в главе 6. Однако более практичным, чем предложенное расширение регулярных языков в сторону элементарных, может

оказаться расширение элементарных языков в сторону регулярных, например, путём расширения сигнатуры языка ограничений алгебраических типов данных предикатами принадлежности терма (несинхронному) регулярному языку. Соответствующий класс индуктивных инвариантов и метод вывода инвариантов для него предложены в следующей главе.

Глава 4. Коллаборативный вывод комбинированных инвариантов

В данной главе предложен метод вывода комбинированных инвариантов. Комбинированными инвариантами (раздел 4.2.1) мы называем индуктивные инварианты программ, выраженные в расширении языка ограничений предикатами принадлежности терма языку из произвольного класса инвариантов. Метод, представленный в разделе 4.2, позволяет модифицировать любой алгоритм вывода инвариантов в языке ограничений, основанный на направляемом контрпримерами уточнении абстракций (CEGAR) [51], таким образом, чтобы этот алгоритм эффективно выводил комбинированные инварианты. Модификация этого метода заключается в коллаборативном взаимообмене информацией с алгоритмом вывода инвариантов в классе, с которым производится комбинация (эта идея описана в разделе 4.1).

4.1 Идея коллаборативного вывода

Для простоты изложения ключевая идея коллаборативного вывода представлена как модификации подхода CEGAR для систем переходов. Как будет показано в следующем разделе, системы дизъюнктов Хорна и программы на произвольных языках программирования являются частным случаем систем переходов.

4.1.1 CEGAR для систем переходов

Пусть $\langle \mathcal{S}, \subseteq, 0, 1, \cap, \cup, \neg \rangle$ — это полная булева решётка, представляющая множества состояний программы.

Определение 20. *Система переходов (программа)* является тройкой $TS = \langle \mathcal{S}, Init, T \rangle$, где $Init \in \mathcal{S}$ — это *начальные состояния*, а функция $T : \mathcal{S} \mapsto \mathcal{S}$ называется *функцией перехода* и имеет следующие свойства:

- T монотонна, т. е. из $s_1 \subseteq s_2$ следует $T(s_1) \subseteq T(s_2)$;
- T аддитивна, т. е. $T(s_1 \cup s_2) = T(s_1) \cup T(s_2)$;
- $T(0) = Init$.

Определение 21. Состояния $s \in \mathcal{S}$ называются *достижимыми* из множества состояний $s' \in \mathcal{S}$, если существует такое $n \geqslant 0$, что $s = T^n(s')$.

Определение 22. *Проблема безопасности* является парой, включающей программу TS и некоторое свойство $Prop \in \mathcal{S}$. Программа называется *безопасной* относительно этого свойства, если для всех n выполняется $T^n(Init) \subseteq Prop$, иначе она называется *небезопасной*.

Безопасность может быть доказана при помощи *(безопасного) индуктивного инварианта* $I \in \mathcal{S}$, для которого должно выполняться следующее:

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

Так как все индуктивные инварианты по определению являются неподвижными точками функции перехода T, то часто при поиске индуктивного инварианта будут рассматриваться именно неподвижные точки.

Теорема 13 (см. [6]). Программа безопасна тогда и только тогда, когда она имеет безопасный индуктивный инвариант.

Для того, чтобы *автоматически выводить* индуктивные инварианты, обычно фиксируется некоторый *класс инвариантов* $\mathcal{I} \subseteq \mathcal{S}$. *Верификатор* — это алгоритм, который по проблеме безопасности возвращает инвариант в классе инвариантов $\mathcal{I}$ в том случае, если программа безопасна, или контрпример в противном случае. Класс инвариантов $\mathcal{I}$ называется *доменом* верификатора. Верификатор может не завершаться, например, в том случае, когда программа безопасна, но в его домене нет инварианта, доказывающего безопасность.

Определение 23. Полную решётку $\mathcal{A} = \langle A, \sqsubseteq, \bot_{\mathcal{A}}, \top_{\mathcal{A}}, \sqcap, \sqcup \rangle$ будем называть *абстрактным доменом*, а её элементы — *абстрактными состояниями*.

Связка Галуа (Galois connection) [100] или *абстракция* — это пара отображений $\langle \alpha, \gamma \rangle$ между частично упорядоченными множествами $\langle \mathcal{S}, \subseteq \rangle$ и $\langle \mathcal{A}, \sqsubseteq \rangle$, для которых выполнено следующее:

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

Абстрактный домен вместе со связкой Галуа однозначно определяет класс инвариантов $\{\gamma(a) \mid a \in \mathcal{A}\}$, который также будет обозначаться как $\mathcal{A}$. В дальнейшем подразумевается, что проверки вида $\gamma(a) \subseteq Prop$ вычислимы.

Определение 24. *Абстрактная функция перехода* $\hat{T} : \mathcal{A} \mapsto \mathcal{A}$ «поднимает» функцию перехода в абстрактный домен, т. е. для всех $a \in \mathcal{A}$ справедливо следующее:

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

Представленная ниже классическая теорема [67] показывает, как абстракции могут быть применены для верификации.

Теорема 14. Пусть дана программа $TS = \langle \mathcal{S}, Init, T \rangle$ и свойство $Prop$. Тогда $\gamma(a)$ является индуктивным инвариантом $\langle TS, Prop \rangle$, если существует элемент $a \in \mathcal{A}$, такой, что выполнено $\alpha(Init) \sqsubseteq a, \quad \hat{T}(a) \sqsubseteq a, \quad \gamma(a) \subseteq Prop$.

Вход: программа TS и свойство $Prop$.
Выход: $SAFE$ и индуктивный инвариант
или $UNSAFE$ и контрпример.

```
1 ⟨α, γ⟩ ← InitialAbstraction()
2 пока true
3     cex, A ← ModelCheck(TS, Prop, ⟨α, γ⟩)
4     если cex пуст то
5         вернуть SAFE(A)
6     если IsFeasible(cex) то
7         вернуть UNSAFE(cex)
8     ⟨α, γ⟩ ← Refine(⟨α, γ⟩, cex)
```

Листинг 4.1 — Подход CEGAR для систем переходов

Псевдокод подхода CEGAR для систем переходов представлен на листинге 4.1. Алгоритм начинается с построения начальной абстракции $\langle \alpha, \gamma \rangle$, например, с помощью тривиальных отображений $\alpha(s) = \bot_{\mathcal{A}}$ и $\gamma(a) = 1$. Далее при помощи абстракции процедура ModelCheck строит по программе конечную последовательность абстрактных состояний $\overline{a} = \langle a_0, \ldots, a_n \rangle$ таких, что:

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

Если для какого-то i имеем $\gamma(a_i) \not\subseteq Prop$, то возвращается *абстрактный контрпример cex* — либо в паре с $A = 0$ (если $i = 0$), либо в паре с $A = \gamma(a_{i-1})$, причём

$\gamma(a_{i-1}) \subseteq Prop$. Если для всех i справедливо $\gamma(a_i) \subseteq Prop$, и на некотором шаге выполняется $\hat{T}(a_n) \sqsubseteq a_n$, то $\gamma(a_n)$ является индуктивным инвариантом, и поэтому ModelCheck возвращает пустой *cex* и при этом $A = \gamma(a_n)$. Понятие абстрактного контрпримера определяется в каждой конкретной реализации CEGAR. Таким образом, возвращаемое процедурой ModelCheck значение удовлетворяет следующему свойству:

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

Если процедура ModelCheck вернула пустой абстрактный контрпример, то программа безопасна и CEGAR возвращает $\gamma(a_n)$ в качестве индуктивного инварианта. В противном случае необходима проверка того, соответствует ли абстрактному контрпримеру какой-либо конкретный контрпример в исходной программе (процедура IsFeasible). Если это так, то CEGAR останавливается и возвращает этот контрпример, а иначе переходит к итеративному уточнению абстракции $\langle\alpha,\gamma\rangle$ для исключения контрпримера *cex* (процедура Refine).

4.1.2 Коллаборативный вывод путём модификации CEGAR

В данном разделе предложен подход к коллаборативному выводу комбинированных инвариантов. Подход основан на коллаборации двух алгоритмов вывода инвариантов и является асимметричным в следующем смысле. Во-первых, требуется, чтобы один из алгоритмов был экземпляром CEGAR, а другой может быть произвольным. Во-вторых, всем процессом управляет основной CEGAR-цикл, многократно вызывая второй алгоритм.

Предлагаемый подход назван CEGAR($\mathcal{O}$), поскольку процесс «коллаборации» можно рассматривать как алгоритм CEGAR, дополненный вызовами некоторого оракула $\mathcal{O}$. Пусть доменами верификаторов являются классы $\mathcal{A}$ и $\mathcal{B}$ соответственно. CEGAR($\mathcal{O}$) позволяет строить инварианты в классе, являющемся теоретико-множественным объединением этих классов.

Определение 25. Для классов состояний $\mathcal{A} \subseteq \mathcal{S}$ и $\mathcal{B} \subseteq \mathcal{S}$ *комбинированный класс* состояний определяется следующим образом:

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

Комбинированным (индуктивным) инвариантом над $\mathcal{A}$ и $\mathcal{B}$ называется индуктивный инвариант в классе $\mathcal{A} \uplus \mathcal{B}$.

При помощи комбинированных инвариантов можно верифицировать больше программ, чем верификаторами для комбинируемых абстрактных доменов по отдельности. Коллаборативный подход объединяет сильные стороны верификаторов для отдельных классов и способен сходиться на многих задачах, где отдельные верификаторы не будут завершаться.

Пример 10 ($ForkJoin$). Рассмотрим трансформацию, которая преобразует параллельные программы следующим образом. На любом шаге трансформация может недетерминированно устранить все потоковые операции, объединив все потоки с главным ($Join$), и перейти к последовательному исполнению (Seq). Если программа заканчивается последовательным кодом, то трансформация вставляет порождение новых потоков ($Fork$), за которым следуют произвольные преобразования потоков. Если в каком-то фрагменте заданной программы встречается объединение потоков после порождения новых, то этот фрагмент не изменяется.

Эта трансформация может быть представлена в виде функциональной программы. Для её описания не нужно рассматривать базовые конструкции языка программирования кроме тех, которые связаны с потоками, поэтому для представления программ может быть использован следующий алгебраический тип данных:

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

Например, терм `Fork(Join(Seq))` представляет программу, которая порождает новые потоки, затем в какой-то момент их объединяет, а затем работает только последовательно.

Одним из свойств описанной трансформации является следующее: если исходная программа состоит из последовательности подряд идущих разветвлений и объединений потоков, то она никогда не может быть преобразована сама в себя. Это свойство вместе с самой трансформацией представлено функциональной программой на листинге 4.2.

Функция `tr` выполняет описанную выше трансформацию над представлением программы алгебраическим типом данных $Prog$, в частности, вводит произвольные преобразования потоков, вызывая функцию `randomTransform`. Функция `ok` проверяет, что программа является последовательностью подряд идущих операций разветвления (`Fork`) и объединения (`Join`) потоков. Утверждение в конце кодирует свойство, которое необходимо проверить.

```
type Prog = Seq | Fork of Prog | Join of Prog
fun randomTransform() : Prog
fun nondet() : bool

fun tr(p : Prog) : Prog =
    match nondet(), p with
    | false, Seq -> Fork(randomTransform())
    | false, Fork(Join(p')) -> Fork(Join(tr(p')))
    | _ -> Join(Seq)

fun ok(p : Prog) : bool =
    match p with
    | Seq -> true
    | Fork(Join(p')) -> ok(p')
    | _ -> false

(* для произвольной программы p : Prog *)
assert (not ok(p) or tr(p) <> p)
```

Листинг 4.2 — Пример функциональной программы с алгебраическими типами данных

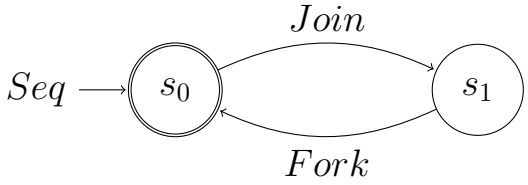
Эта программа безопасна относительно заданного свойства, однако не имеет индуктивных инвариантов, выразимых в классах Elem или Reg для функций `ok` и `tr`. Вместе с тем у неё есть комбинированные инварианты в Elem ⊎ Reg. Так, для любых программ $p, t : Prog$ функция `ok`(p) возвращает `true`, если справедливо следующее утверждение:

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

Если `tr`$(p) = t$, тогда следующая формула представляет собой индуктивный инвариант для функции `tr`:

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

где $t \notin \mathcal{E}$ означает, что АТД-терм t *не* содержится в языке $\mathcal{E}$ следующего автомата над деревьями:



Исходные параметры: верификатор $\mathcal{O}$ над доменом $\mathcal{B}$.
Вход: программа TS и свойство $Prop$.
Выход: $SAFE$ и комбинированный инвариант в $\mathcal{A} \uplus \mathcal{B}$
или $UNSAFE$ и контрпример cex.

```
1  ⟨α, γ⟩ ← Initial()
2  A ← 0
3  пока true
4      асинхронно вызвать Collaborate (TS, Prop, ⟨α, γ⟩ , A)
5      cex, A ← ModelCheck(TS, Prop, ⟨α, γ⟩)
6      если cex пуст то
7          вернуть SAFE(A)
8      если IsFeasible(cex) то
9          вернуть UNSAFE(cex)
10     ⟨α, γ⟩ ← Refine(⟨α, γ⟩ , cex)
```

Листинг 4.3 — Основной цикл алгоритма CEGAR($\mathcal{O}$)

Описание алгоритма CEGAR($\mathcal{O}$). Предлагаемый коллаборативный подход представлен на листинге 4.3. Алгоритм работает аналогично классическому CEGAR, представленному в разделе 4.1.1, но дополнительно в начале каждой итерации он асинхронно опрашивает коллаборирующий верификатор $\mathcal{O}$ путём вызова процедуры Collaborate (строка 4). Вызовы делаются асинхронно, чтобы предотвратить зацикливание алгоритма.

Процедура **Collaborate** представлена на листинге 4.4. По исходной проблеме безопасности, текущей абстракции и множеству состояний $A = \gamma(a)$ для некоторого $a \in \mathcal{A}$ она строит новую *остаточную* систему переходов:

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

Исходные параметры: Верификатор $\mathcal{O}$ над доменом $\mathcal{B}$.
Вход: Программа $TS = \langle \mathcal{S}, Init, T \rangle$, свойство $Prop$, абстракция $\langle \alpha, \gamma \rangle$, множество состояний A, такое что $A = 0$ или $Init \subseteq A \subseteq Prop$.

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 **если** *cex пуст* **то**
4 **выйти и вернуть** $SAFE(A \cup B)$
5 $\widehat{cex} \leftarrow \textsc{RecoverCex}(TS, Prop, \langle \alpha, \gamma \rangle, A, cex)$
6 $\langle \alpha, \gamma \rangle \leftarrow \textsc{Refine}(\langle \alpha, \gamma \rangle, \widehat{cex})$

Листинг 4.4 — Процедура Collaborate

Затем безопасность остаточной системы проверяется коллаборирующим верификатором $\mathcal{O}$. На листинге $A \setminus B$ является сокращением для $A \cap \neg B$. Процедура Collaborate перезаписывает абстракцию, используемую в CEGAR (стр. 6): абстракция $\langle \alpha, \gamma \rangle$ является глобальной и разделяется между двумя процедурами.

Остаточная система устроена следующим образом. Её состояния в исходной системе достижимы из состояний, нарушающих индуктивность A. В частности, её начальные состояния $Init'$ — это $T(A) \setminus A$, т. е. образ неиндуктивных состояний. Состояния, достижимые за один шаг в остаточной системе $T'(Init') = T(T(A) \setminus A) \setminus A$ — это T-образ неиндуктивных состояний.

Основная идея алгоритма CEGAR($\mathcal{O}$) заключается в том, чтобы использовать дополнительный верификатор $\mathcal{O}$ для *ослабления* неиндуктивного множества состояний A до некоторой неподвижной точки в комбинированном классе. Если второй верификатор находит индуктивный инвариант остаточной системы, т. е. некую индуктивную аппроксимацию B неиндуктивных состояний, то $A \cup B$ будет индуктивным инвариантом исходной системы. Иными словами, путём построения остаточной системы алгоритм берёт безопасную, но неиндуктивную часть текущего кандидата в инварианты и передаёт её сотрудничающему верификатору, чтобы он дополнил её до неподвижной точки, т. е. индуктивного инварианта.

Современные подходы к выводу индуктивных инвариантов программ, такие как IC3/PDR (который можно рассматривать как усложнённый вариант CEGAR), монотонно *усиливают* кандидат в инварианты A до тех пор, пока

он не станет индуктивным. В силу этого проблемой таких подходов является выбор стратегии усиления [101]: из-за слишком резкого усиления могут быть пропущены нужные неподвижные точки, в то время как из-за медленного усиления алгоритм может сходиться к неподвижной точке слишком долго или вовсе расходиться.

Так как предлагаемый подход не является монотонным, он позволяет строить инварианты, невыводимые верификаторами по отдельности. Кроме того, он (эвристически) может ускорить построение инварианта даже в том случае, если один из верификаторов может вывести его самостоятельно (эта гипотеза проверена в разделе 6.4.2). Вероятность пропуска неподвижных точек из-за слишком резкого усиления уменьшается: даже если первый верификатор чрезмерно усиливает кандидата в инварианты, второй верификатор всё равно может обнаружить более слабую неподвижную точку.

Таким образом, если второй верификатор $\mathcal{O}$ останавливается и возвращает индуктивный инвариант B, то Collaborate возвращает комбинированный инвариант $A \cup B$. Если $\mathcal{O}$ возвращает конкретный контрпример cex к остаточной системе, то Collaborate строит по нему абстрактный контрпример $\widehat{cex}$ к исходной системе и далее действует как обычный CEGAR, уточняя домен с помощью $\widehat{cex}$.

Заметим, что множеств состояний A и B самих по себе недостаточно для доказательства безопасности исходной системы переходов. Другими словами, коллаборация осуществляется путём делегирования более *простых* проблем верификатору $\mathcal{O}$, решение которых даёт только *часть* ответа на исходную задачу.

Лемма 5. Если процедура Collaborate $(TS, Prop, \langle \alpha, \gamma \rangle, a)$ останавливается с результатом $SAFE(A \cup B)$ (стр. 4), то $A \cup B$ является комбинированным инвариантом проблемы $\langle TS, Prop \rangle$.

Доказательство. Докажем, что $A \cup B$ является индуктивным инвариантом, показав, что для него выполняются все три признака индуктивных инвариантов из определения 22: в этом множестве содержатся все начальные состояния, оно сохраняет отношение перехода и оно является подмножеством свойства.

Начальные состояния. Из инварианта (4.2) алгоритма CEGAR получаем следующие случаи. Либо $Init \subseteq A \subseteq A \cup B$, что и требовалось доказать. Либо $A = 0$, и тогда по определению $T(0)$, $Init = T(0) \setminus 0 = T(A) \setminus A \subseteq B \subseteq A \cup B$.

Сохранение отношения перехода. В силу корректности верификатора $\mathcal{O}$ имеем, что множество B является индуктивным инвариантом $(\langle \mathcal{S}, Init', T' \rangle, Prop)$, т. е. $T(A) \setminus A \subseteq B$ (определение $Init'$) и $T(B) \setminus A \subseteq B$ (определение T'). Поэтому $(T(A) \cup T(B)) \setminus A \subseteq B$, из чего следует $T(A) \cup T(B) \subseteq A \cup B$, поэтому, так как функция T аддитивна, имеем $T(A \cup B) \subseteq A \cup B$.

Подмножество свойства. Справедливость $A \subseteq Prop$ следует из инварианта (4.2) алгоритма CEGAR и $B \subseteq Prop$ по предположению корректности алгоритма $\mathcal{O}$. Следовательно, имеем $A \cup B \subseteq Prop$. □

Контрпримерами для остаточной системы являются трассы, которые нарушают индуктивность текущего кандидата в инварианты A. *Конкретный* контрпример к безопасности остаточной системы (*cex* на стр. 2 с листинга 4.4) соответствует некоторому *абстрактному* контрпримеру исходной системы. Поэтому CEGAR($\mathcal{O}$) параметризован процедурой RecoverCex, которая восстанавливает абстрактный контрпример к исходной системе по контрпримеру к остаточной системе (стр. 5). В следующем разделе 4.2 предложена такая процедура для программ, представленных системами дизъюнктов Хорна, и контрпримеров, представленных деревьями опровержений.

Процедура RecoverCex должна удовлетворять следующему ограничению.

Ограничение 1. RecoverCex $(TS, Prop, \langle \alpha, \gamma \rangle, a, cex)$ возвращает абстрактный контрпример к системе переходов $\langle TS, Prop \rangle$ относительно абстракции $\langle \alpha, \gamma \rangle$.

Теорема 15. Если верификатор $\mathcal{O}$ корректен, то верификатор CEGAR($\mathcal{O}$) тоже корректен.

Доказательство. Справедливость этой теоремы непосредственно следует из корректности исходного CEGAR [51], леммы 5 и ограничения 1. □

Теорема 16. Если верно, что либо CEGAR, либо верификатор $\mathcal{O}$ завершаются на системе $\langle TS, Prop \rangle$, то CEGAR($\mathcal{O}$) также завершается на этой же системе.

Доказательство. Если верификатор $\mathcal{O}$ завершается, то завершается и первый вызов процедуры Collaborate $(TS, Prop, \langle \alpha, \gamma \rangle, 0)$, так как $Init' = T(0) \setminus 0 = Init$ и $T' = \lambda B.(T(B) \setminus 0) = T$. Если CEGAR завершается, то завершается и CEGAR($\mathcal{O}$), так как вызов Collaborate является асинхронным. □

4.2 Коллаборативный вывод инвариантов

В данном разделе весь подход в целом представлен как инстанциация алгоритма CEGAR($\mathcal{O}$) из прошлого раздела для систем дизъюнктов Хорна над АТД.

Итоговый подход обладает следующими двумя свойствами. Во-первых, он позволяет выводить индуктивные инварианты, выраженные в языке первого порядка над АТД, обогащённом ограничениями на принадлежность термов языкам деревьев $\overline{x} \in L$. Во-вторых, подход позволяет расширить Хорн-решатели запросами к решателям для логики первого порядка, например, основанным на насыщении [96], а также инструментам поиска конечных моделей [90; 91].

Прежде всего, определим, как будет выражаться класс комбинированных инвариантов.

4.2.1 Комбинированные инварианты

Определение 26. Для каждого языка деревьев $\mathbf{L} \subseteq |\mathcal{H}|_{\sigma_1} \times \cdots \times |\mathcal{H}|_{\sigma_m}$ определим предикатный символ принадлежности языку "$\in \mathbf{L}$" с арностью $\sigma_1 \times \cdots \times \sigma_m$. *Ограничение принадлежности* — это атомарная формула с предикатным символом принадлежности языку. Его семантика определяется расширением семантики Эрбрана $\mathcal{H}$ следующим образом: $\mathcal{H}(\in \mathbf{L}) = \mathbf{L}$. Язык ограничений АТД, расширенный такими атомами, называется *языком первого порядка с ограничениями принадлежности*. Этот язык задаёт класс инвариантов обозначаемый ElemReg и абстрактный домен функций из предикатов $\mathcal{R}$ в формулы языка с поэлементными операциями.

Пример 11. Функциональная программа из примера 10 соответствует следующей системе дизъюнктов Хорна:

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

Она безопасна, но не имеет ни Reg-, ни Elem-инвариантов. Однако, у неё есть следующий ElemReg-инвариант:

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

где $\mathcal{E}$ — это язык деревьев, задаваемый автоматом из примера 10, а $\overline{\mathcal{E}}$ является его дополнением.

4.2.2 Система дизъюнктов Хорна как система переходов

Зададим полную булеву решетку конкретных состояний $\langle \mathcal{S}, \subseteq, 0, 1, \cap, \cup, \neg \rangle$. Определим $\mathcal{S}$ как множество всех отображений из символов $P \in \mathcal{R}$ в подмножества $|\mathcal{H}|_P$. Определим также следующие операции:

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

Система дизъюнктов Хорна $\mathcal{P}$ задаёт систему переходов $\langle \mathcal{S}, Init, T \rangle$:

$$Init \triangleq T\,(0)$$
$$T\,(s)\,(P) \triangleq \left\{\bar{t} \mid \left(B \to P(\bar{t})\right) \text{— замкнутый экземпляр некоторого } C \in \mathcal{P}, s \models B\right\}$$

Без потери общности предположим, что система дизъюнктов Хорна $\mathcal{P}$ имеет единственный предикат для запроса Q, т. е. далее будем рассматривать только системы, полученные следующим образом:

$$\mathcal{P}' \triangleq rules\,(\mathcal{P}) \cup \{body\,(C)\,(\overline{x}) \to Q(\overline{x}) \mid C \text{ является запросом } \mathcal{P}\} \cup \{Q(\overline{x}) \to \bot\}.$$

Система дизъюнктов определяет свойство для системы переходов так: $Prop(Q) \triangleq \bot$ и для каждого $P \in \mathcal{R}$, $Prop(P) \triangleq \top$.

Свойство 1. Система дизъюнктов Хорна $\mathcal{P}$ выполнима, если соответствующая система переходов $\langle \mathcal{S}, Init, T \rangle$ безопасна относительно $Prop$.

4.2.3 Порождение остаточной системы

Процедура Collaborate начинает с построения остаточной системы $\langle T\,(A) \cap \neg A, \lambda B.\,(T\,(B) \cap \neg A) \rangle$. Эта система передаётся коллаборирующему

верификатору. Процедура ResidualCHCs, представленная на листинге 4.5, строит систему, эквивалентную такой остаточной системе, преобразуя исходную систему дизъюнктов Хорна $\mathcal{P}$ в два шага. Она принимает на вход исходную систему дизъюнктов $\mathcal{P}$, элемент абстрактного домена a и функцию из предикатных символов в формулы языка ElemReg.

Вход: Система дизъюнктов Хорна $\mathcal{P}$, функция из предикатных символов в формулы a.
Выход: Остаточная Хорн-система $\mathcal{P}'$.

1 $\Phi \leftarrow \mathcal{P}$ с атомами $P(\bar{t})$ заменёнными на $a(P)(\bar{t}) \vee P(\bar{t})$
2 **вернуть** (Φ)

Листинг 4.5 — Алгоритм построения остаточной Хорн-системы ResidualCHCs

Процедура заменяет каждый атом $P(t_1, \ldots, t_m)$ в голове и теле каждого дизъюнкта Хорн-системы дизъюнкцией $a(P)(t_1, \ldots, t_m) \vee P(t_1, \ldots, t_m)$ (стр. 1), и затем приводит его в КНФ. Например, дизъюнкт

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

сначала превратится в формулу

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

которая после преобразования в КНФ (стр. 2) будет разбита на следующие дизъюнкты:

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

Таким образом, в результате преобразования в КНФ мы получаем систему дизъюнктов, которая семантически соответствует остаточной системе из прошлого раздела.

Пример 12. Возьмём абстрактное состояние $a(tr)(p, t) \equiv \neg(p = t) \vee t = Join(Seq)$, $a(ok)(p) \equiv p = Seq$ и систему из примера 11. Процедура

ResidualCHCs сначала даст следующую формулу:

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

Эта формула может быть упрощена до следующей системы дизъюнктов:

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

4.2.4 CEGAR($\mathcal{O}$) для дизъюнктов: восстановление контрпримеров

В данном разделе представлена процедура построения абстрактного контрпримера исходной системы по конкретному контрпримеру для остаточной системы, получаемой как $\mathcal{P}' = \text{ResidualCHCs}(\mathcal{P}, a)$. Иными словами, представлена инстанциация процедуры RecoverCex из листинга 4.4.

Абстрактные контрпримеры. Определим абстрактный контрпример к системе дизъюнктов Хорна как дерево опровержений, некоторые листья которого могут быть абстрактными состояниями. Для формально определения введём трансформацию дизъюнктов $Q(\mathcal{P}, a)$, которая для каждого предиката $P \in \mathcal{R}$ добавляет к системе $\mathcal{P}$ новые дизъюнкты $a(P)(\overline{x}) \to P(\overline{x})$.

Определение 27. *Абстрактный контрпример* для Хорн-системы $\mathcal{P}$ относительно абстрактного состояния a является деревом опровержений Хорн-системы $Q(\mathcal{P}, a)$.

Пусть T — это дерево опровержений для системы $\mathcal{P}' = \text{ResidualCHCs}(\mathcal{P}, a)$. Далее приведена рекурсивная процедура построения дерева опровержений T' для системы $\mathcal{P}'' = Q(\mathcal{P}, a)$ по дереву T.

База рекурсии. Пусть T состоит из единственной вершины $\langle C, \Phi \rangle$, где $C \in \mathcal{P}'$. Поскольку $\Phi = body\,(C)$ — формула без предикатов, то C имеет следующий вид:

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

Построим T' как вершину $\langle C', \Phi' \rangle$, где C' определим как $\varphi \wedge P_1(\overline{x}_1) \wedge \ldots \wedge P_n(\overline{x}_n) \to P(\overline{x})$ и $\Phi' \equiv \varphi \wedge a(P_1)(\overline{x}_1) \wedge \ldots \wedge a(P_n)(\overline{x}_n)$ с n дочерними листьями $\langle C'_i, a(P_i)(\overline{x}_i) \rangle$, где $C'_i \equiv a(P_i)(\overline{x}_i) \to P_i(\overline{x}_i)$. Определение дерева опровержения для T' тривиально выполняется. Заметим, что $\mathcal{H} \models \Phi \to \Phi'$.

Итерация рекурсии. Пусть T — это узел $\langle C, \Phi \rangle$ с детьми $\langle C_1, \Phi_1 \rangle, \ldots, \langle C_n, \Phi_n \rangle$, все $C_i \in rules\,(P_i)$ из $\mathcal{P}'$ и

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

Благодаря итерации рекурсии у нас уже есть соответствующие им узлы $\langle C'_1, \Phi'_1 \rangle, \ldots, \langle C'_n, \Phi'_n \rangle$. Поэтому определим следующее:

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

Для каждого предиката R_j добавим следующих потомков — $\langle C'_{n+j}, a(R_j)(\overline{y}_j) \rangle$, где $C'_{n+j} \equiv a(R_i)(\overline{y}_i) \to R_j(\overline{y}_j)$.

Для каждого i имеем $\mathcal{H} \models \Phi_i \to \Phi'_i$ по индукции, поэтому для их конъюнкции имеем $\mathcal{H} \models \Phi \to \Phi'$.

В конце концов рекурсия придёт к корню дерева T, некоторой вершине $\langle C, \Phi \rangle$, где C — это запрос из системы $\mathcal{P}'$. Для него рекурсивно построено дерево T' с корнем $\langle C', \Phi' \rangle$. Также по индукции имеем: $\mathcal{H} \models \Phi \to \Phi'$. Поскольку Φ является выполнимой Σ-формулой, то Φ' тоже выполнима. Таким образом, T' является деревом опровержения системы $\mathcal{P}''$.

Свойство 2. Процедура RecoverCex имеет линейную сложность от числа узлов входного дерева опровержения.

4.2.5 Инстанцирование подхода в рамках IC3/PDR

Предложенный подход может быть реализован в рамках алгоритма IC3/PDR [55], который успешно применяется вывода индуктивных инвариантов в разных теориях [24], если рассмотреть его как сложную версию алгоритма CEGAR.

Приведённый алгоритм позволяет выводить комбинированные инварианты в классе ElemUnionReg, т. е. индуктивные инварианты, выразимые формулами вида $\varphi(\overline{x}) \vee \overline{x} \in L$, где φ — формула первого порядка над АТД, а L — язык деревьев. Реализация подхода в рамках IC3/PDR как сложной инстанциации CEGAR может быть обобщена для автоматического вывода инвариантов в полном бесквантором фрагменте ElemReg, состоящем из формул следующего вида:

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

IC3/PDR представляет абстрактное состояние в виде конъюнкции формул (называемых *леммами*). Другими словами, в процедуре ResidualCHCs$(\mathcal{P}, a)$ (см. раздел 4.2.3) функция a отображает каждый неинтерпретированный символ P в некоторую конъюнкцию $\bigwedge_i \varphi_i$. Обобщение подхода получается путём замены в этой процедуре применений неинтерпретированного предикатного символа P на *конъюнкции дизъюнкций* $\bigwedge_i (\varphi_i(\overline{t}) \vee L_i(\overline{t}))$ с новыми предикатными символами L_i. Таким образом, будут выводиться индуктивные инварианты вида 4.5 выше.

4.3 Выводы

Предложенный класс комбинированных инвариантов, построенный на регулярных инвариантах, позволяет выражать как классические символьные инварианты, так и сложные рекурсивные отношения. Тем самым, предложенный класс инвариантов является достаточно выразительным и может использовать на практике. Кроме того, для него предложен эффективный метод вывода инвариантов, позволяющий путём небольшой модификации повторно использовать существующие эффективные алгоритмы вывода инвариантов для комбинируемых классов. В следующей главе на теоретическом уровне сопоставлены существующие и предложенные классов индуктивных инвариантов.

Глава 5. Теоретическое сравнение классов индуктивных инвариантов

В данной главе приведено теоретическое сопоставление существующих и предложенных классов индуктивных инвариантов для программ с алгебраическими типами данных. Рассмотрены только те классы, известные из литературы, для которых существуют полностью автоматические методы вывода инвариантов: элементарные инварианты (Elem, выводятся с помощью инструментов Spacer [24] и HoIce [27]), элементарные инварианты с ограничениями размера термов (SizeElem, выводятся с помощью инструмента Eldarica [26]), регулярные инварианты (Reg, выводятся с помощью инструмента RCHC [28], а также методом из главы 2), синхронные регулярные инварианты (Reg_{+}, $\text{Reg}_{\times}$, выводятся с помощью инструмента RCHC [28], а также методом из главы 3) и комбинированные инварианты (ElemReg, выводятся методом из главы 4).

Сопоставление выполнено на базе свойств, которые являются ключевыми для классов инвариантов: замкнутость относительно булевых операций, разрешимость задачи принадлежности кортежа термов инварианту, разрешимость проверки инварианта на пустоту (раздел 5.1), выразительная сила (раздел 5.2). Результаты теоретического сравнения приведены в таблицах 5.1 и 5.2. В разделе 5.3 представлен обзор альтернативных рассмотренным способов представления бесконечных множеств термов, основанных на обобщениях автоматов над деревьями, которые могут послужить в качестве классов индуктивных инвариантов программ в будущем.

5.1 Замкнутость классов относительно булевых операций и разрешимость операций

Свойства замкнутости и разрешимости для рассмотренных классов сведены в таблицу 5.1. Сноска в каждой ячейке таблица отсылает к соответствующей теореме; отсутствие сноски свидетельствует об очевидности утверждаемого в ячейке факта. Например, замкнутость классов Elem, SizeElem и ElemReg относительно булевых операций очевидна, т. к. они синтаксически строятся как языки первого порядка с соответствующими операциями.

Таблица 5.1 — Теоретическое сравнение классов индуктивных инвариантов

Класс / Свойство	Elem	SizeElem	Reg	Reg_+	$\text{Reg}_\times$	ElemReg
Замкнут по $\cap$	Да	Да	Да[1]	Да[2]	Да[2]	Да
Замкнут по $\cup$	Да	Да	Да[1]	Да[2]	Да[2]	Да
Замкнут по $\setminus$	Да	Да	Да[1]	Да[2]	Да[2]	Да
Разрешимо $\bar{t} \in I$	Да[3]	Да[4]	Да[5]	Да[7]	Да[9]	Да[10]
Разрешимо $I = \varnothing$	Да[3]	Да[4]	Да[6]	Да[8]	Да[9]	Да[10]
Выразимы рекурсивные отношения	Нет	Частично	Да	Да	Да	Да
Выразимы синхронные отношения	Да	Да	Нет	Частично	Да	Да

[1] см. [35, свойство 3.2.9]
[2] см. раздел 3.1.2
[3] см. [95]
[4] см. [102]
[5] см. [35, разд. 3.2.1 и теор. 1.7.2]
[6] см. [35, разд. 3.2.1 и теор. 1.7.4]
[7] см. [35, опр. 3.2.1 и теор. 1.7.2]
[8] см. [35, опр. 3.2.1 и теор. 1.7.4]
[9] см. раздел 3.1.3
[10] см. [103, следствие 2]

5.2 Сравнение выразительности классов инвариантов

Результаты сравнения выразительности классов инвариантов представлены в таблице 5.2. Поскольку некоторые классы построены как синтаксические расширения других классов (например, Reg_+ и $\text{Reg}_\times$ как расширяют Reg), а некоторые классы синтаксически различаются очень сильно (например, Reg и Elem), взаимосвязи между представляемыми ими множествами не очевидны. Разграничить классы инвариантов важно для проведения анализа алгоритмов вывода инвариантов, в частности, для понимания границ их применимости. Если инварианты проблем какого-то вида не лежат в классе инвариантов, выводимых данным алгоритмом, то этот алгоритм не будет завершаться на проблемах данного вида. Поэтому эти взаимосвязи важны и представлены в таблице 5.2.

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

Таблица 5.2 — Теоретическое сравнение выразительности классов индуктивных инвариантов

Класс	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] $lr \in \text{Elem} \setminus \text{Reg}_\times$ (лемма 7)
[2] $even \in \text{Reg} \setminus \text{SizeElem}$ (теор. 21)
[3] см. теор. 17
[4] $\text{Reg} \subseteq \text{Reg}_+$ [35, свойство 3.2.6]
[5] $\text{Reg}_+ \subseteq \text{Reg}_\times$ [28, теор. 11]

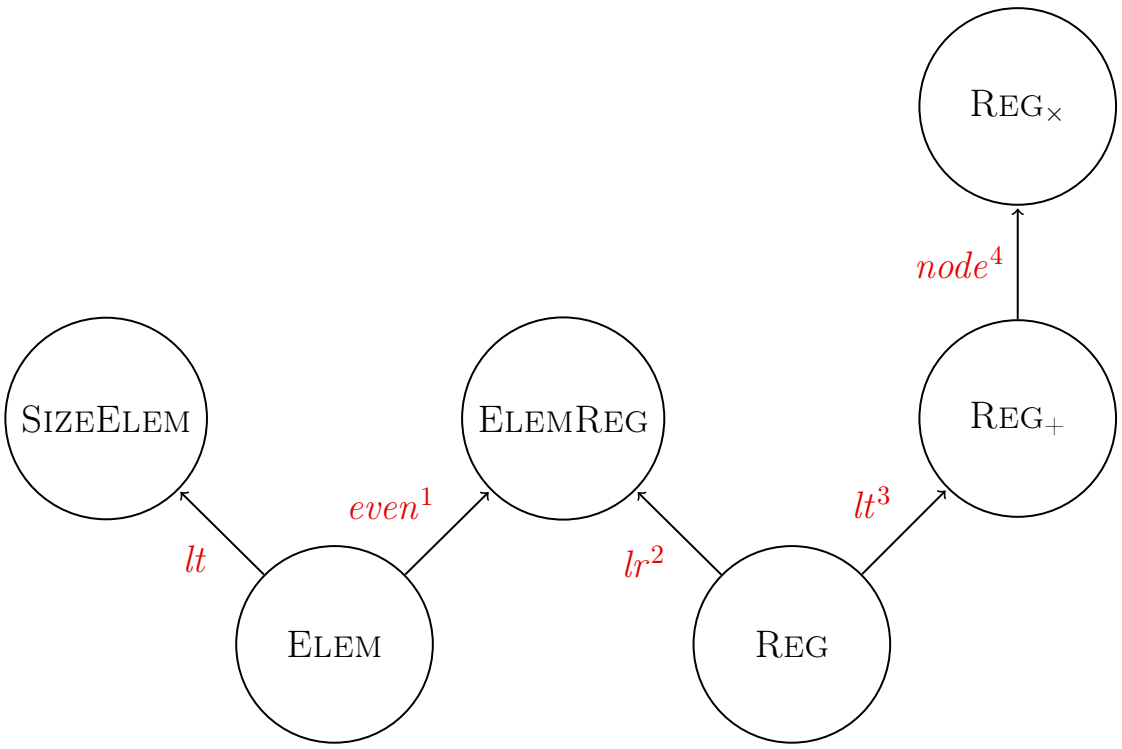


Рисунок 5.1 — Связи включения между классами индуктивных инвариантов над АТД.

Для класса A в строке и класса B в столбце в соответствующей им ячейке находится сноска, а также либо символ $\varnothing$, либо ∞, либо название некоторой системы дизъюнктов из данной работы. Каждую ячейку следует читать как ответ на вопрос: «Что находится в классе $A \setminus B$?» Если в ячейке находится $\varnothing$, значит $A \setminus B = \varnothing$. Если в ячейке находится ∞, значит $B \subseteq A$. Наконец, если в

ячейке находится название системы $\mathcal{P}$, значит, классы A и B несравнимы, т. е. $\mathcal{P} \in A \setminus B \neq \varnothing$ и при этом $B \setminus A \neq \varnothing$. Сноска отсылает к соответствующей теореме, представленной в данной работе. Отсутствие сноски означает очевидность приводимого факта. Например, в ячейке SizeElem $\setminus$ Elem находится ∞ без сноски, т. к. класс SizeElem является синтаксическим расширением класса Elem, следовательно, включает, как минимум, те же инварианты.

На рисунке 5.1 для удобства отдельно представлены связи включения между классами инвариантов. Ребро, ведущее из класса A в класс B с меткой $\mathcal{P}$ означает, что $A \subsetneq B$ и $\mathcal{P} \in B \setminus A$.

5.2.1 Невыразимость в синхронных языках

Пример 13 (*node*). Рассмотрим следующее множество термов над алгебраическим типом бинарных деревьев $Tree ::= left \,|\, node(Tree, Tree)$:

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

Этот пример позволяет разделить классы синхронных регулярных инвариантов с полной и стандартной свёртками, как показывает следующая лемма.

Лемма 6. Существуют синхронные регулярные инварианты с полной свёрткой, которые невыразимы при помощи только стандартной свёртки, т. е. $node \in \text{Reg}_{\times} \setminus \text{Reg}_{+}$.

Доказательство. Факт $node \in \text{Reg}_{\times}$ следует из применения теоремы 11 к языку равенства двух термов и линейному шаблону $\langle Node(y, z), y, z\rangle$. Справедливость $node \notin \text{Reg}_{+}$ показана в [35, упр. 3.2] применением леммы о «накачке» для языков автоматов над деревьями к языку $node$. □

Пример 14 (*lr*). Рассмотрим следующее множество термов над алгебраическим типом бинарных деревьев $Tree ::= left \,|\, node(Tree, Tree)$:

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

Это множество лежит в классе элементарных инвариантов, однако не может быть выражено никаким синхронным автоматов над деревьями даже с полной синхронизацией, как показывает следующая лемма.

Лемма 7. Существуют элементарные инварианты, которые не являются инвариантами, выразимыми регулярно с полной синхронизацией, т. е. $lr \notin \text{Reg}_{\times}$.

Доказательство. В [35, упр. 1.4] применением леммы о «накачке» к языку lr показано, что $lr \notin \text{Reg}$. По лемме 4, из этого следует $lr \notin \text{Reg}_{\times}$. □

5.2.2 Невыразимость в комбинированных языках

Теорема 17. Пересечение классов SizeElem и Reg_{+} не лежит в классе ElemReg, т. е. $lt \in \text{SizeElem}$, $lt \in \text{Reg}_{+}$, $lt \notin \text{ElemReg}$

Доказательство. Множество lt выражается следующей SizeElem-формулой:

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

Истинность утверждения $lt \in \text{Reg}_{+}$ была доказана в примере 8.

Покажем теперь, что lt не лежит в классе ElemReg. Заметим, что алгебраический тип целых чисел Пеано изоморфмен натуральным числам (с нулём). Кроме того, формулы, представляющие множества из класса ElemReg, изоморфны формулам в сигнатуре расширенной арифметики Пресбургера без сложения и порядка. Рассмотрим эту сигнатуру $\Sigma = \langle \Sigma_S, \Sigma_F, \Sigma_P \rangle$, включающую единственный сорт натуральных чисел ($\Sigma_S = \{\mathbb{N}\}$), константа 0 и единственный функциональный символ следования s ($s(x)$ интерпретируется как $x + 1$, $\Sigma_F = \{0, s\}$), а также предикатные символы равенства и делимости на все константы ($c \mid x$ интерпретируется как x делится на c, $\Sigma_P = \{=\} \cup \{c \mid \_, c \in \mathbb{N}\}$). Поскольку множество lt представляет отношение строгого порядка, для доказательства исходного утверждения необходимо показать, что стандартное отношение порядка на натуральных числах невыразимо в теории стандартной модели $\mathcal{N}$ сигнатуры Σ.

Докажем это утверждение, расширив доказательство [104, разд. 2] для арифметики с аналогичной сигнатурой без предикатов делимости. Рассмотрим модель $\mathcal{M} = (\mathbb{N} \cup \mathbb{N}^*, s, c \mid \_)$, где множество $\mathbb{N}^*$ определено как множество символов $\{n^* \mid n \in \mathbb{N}\}$; $c \mid n$ и $c \mid n^*$ выполняются только когда c делит n; функция следования определена следующим образом:

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

Модель $\mathcal{M}$ является элементарным расширением модели $\mathcal{N}$, поэтому если некоторая формула $\psi(x, y)$ определяет линейный порядок на $\mathcal{N}$, она определяет

линейный порядок и на $\mathcal{M}$. Заметим, что следующее отображение σ является автоморфизмом модели $\mathcal{M}$:

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

Из того факта, что σ — автоморфизм модели $\mathcal{M}$, следует, что для любых $x, y \in \mathbb{N} \cup \mathbb{N}^*$ верно, что $\mathcal{M} \models \psi(x, y) \Leftrightarrow \mathcal{M} \models \psi(\sigma(x), \sigma(y))$. Поскольку формула ψ по предположению выражает линейный порядок, без ограничений общности допустим, что $\mathcal{M} \models \psi(0, 0^*)$, но тогда, применив автоморфизм σ, получим, что $\mathcal{M} \models \psi(0^*, 0)$, что противоречит аксиомам порядка. Следовательно, никакая формула данной сигнатуры не может представлять линейный порядок. Из этого следует, что *lt* не лежит в классе ElemReg. □

5.2.3 Невыразимость в элементарных языках

В данном разделе представлены леммы о «накачке» для языков первого порядка — языка ограничений и языка ограничений, расширенного ограничениями на размер термов.

Первые леммы о «накачке» возникли в теории формальных языков [105] в связи с конечными автоматами и контекстно-свободными грамматиками. В общем виде любая лемма о «накачке» должна показывать, что для всех языков в некотором классе (например, регулярных или контекстно-свободных языков) любое достаточно большое слово может быть «накачано». Иными словами, некоторые части слова могут быть неограниченно увеличены, и «накачанное» слово при этом останется в языке. Леммы о «накачке» полезны для доказательства невыразимости инварианта в некотором классе: предположив, что инвариант принадлежит классу, можно применить специализированную лемму о «накачке» для этого класса и получить некоторое «накачанное» множество. Если оно не может быть индуктивным инвариантом, то получено противоречие, следовательно, инвариант невыразим в данном классе.

Для формулировки лемм о «накачке» в начале определим следующее расширение языка ограничений, обозначаемое Elem$*$, которое допускает устранение кванторов. Для каждого АТД $\langle \sigma, C \rangle$ и каждого конструктора $f \in C$, имеющего арность $\sigma_1 \times \cdots \times \sigma_n \to \sigma$ для некоторых сортов $\sigma_1, \ldots, \sigma_n$, введём *селекторы* $g_i \in S$ с арностью $\sigma \to \sigma_i$ для каждого $i \leqslant n$ со стандартной семантикой, заданной так: $g_i(f(t_1, \ldots, t_n)) \triangleq t_i$.

Теорема 18 (см. [95]). Всякая ELEM-формула эквивалентна некоторой бескванторной ELEM$_*$-формуле.

Дадим несколько вспомогательных определений.

Определение 28. *Высоту замкнутого терма* определим индуктивно следующим образом:

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

Будем называть *путём* (возможно пустую) последовательность селекторов $s \triangleq S_1 \ldots S_n$, где для каждого i от 1 до n, S_i имеет сорт $\sigma_i \to \sigma_{i-1}$. Для терма t сорта σ_n пусть $s\,(t) \triangleq S_1(\ldots(S_n(t))\ldots)$. Для замкнутого терма g переопределим $s\,(g)$ как вычисленный подтерм g в s. В дальнейшем будем обозначать пути прописными буквами p, q, r, s.

Мы говорим, что два пути p и q *перекрываются*, если один из них является суффиксом другого. Для попарно неперекрывающихся путей $p_1, \ldots, p_n$ с помощью записи $t[p_1 \leftarrow u_1, \ldots, p_n \leftarrow u_n]$ обозначим терм, полученный одновременной заменой в t подтермов $p_i\,(t)$ на термы u_i. Для конечной последовательности попарно различных путей $P = (p_1, \ldots, p_n)$ и некоторого множества термов $U = (u_1, \ldots, u_n)$ мы переопределяем обозначения и пишем $t[P \leftarrow U]$ вместо $t[p_1 \leftarrow u_1, \ldots, p_n \leftarrow u_n]$, а также $t[P \leftarrow t]$ вместо $t[p_1 \leftarrow t, \ldots, p_n \leftarrow t]$.

Теперь определим множество путей, которое будет «накачиваться».

Определение 29. Терм t является *листовым термом* сорта σ, если это конструктор без параметров, или $t = c(t_1, \ldots, t_n)$, где все t_i являются листовыми термами, а t не содержит никаких собственных подтермов сорта σ. Для замкнутого терма g и сорта σ определим $leaves_\sigma(g) \triangleq \{p \mid p\,(g)$ — листовой терм сорта $\sigma\}$.

Лемма 8 (Лемма о «накачке» для класса ELEM). Для любого элементарного языка n-кортежей $\mathbf{L}$ существует константа $K > 0$ такая, что:

– для каждого кортежа термов $\langle g_1, \ldots, g_n\rangle \in \mathbf{L}$,
– для любого i такого, что $\mathcal{H}eight(g_i) > K$,
– для всех бесконечных сортов $\sigma \in \Sigma_S$ и
– для всех путей p в терме g с глубиной большей K

– существует набор конечных последовательностей путей P_j,
– где $p \in P_i$ и для всех $p_1, p_2 \in \bigcup_j P_j$ верно, что $p_1(g) = p_2(g)$,
– и найдётся константа $N \geqslant 0$ такая, что
– для любого терма t сорта σ с $\mathcal{H}eight(t) > N$ имеет место следующее:

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

Доказательство. Доказательство приведено в работе [37]. □

Фактически, лемма 8 утверждает, что для достаточно больших кортежей термов можно взять любой из самых глубоких подтермов, заменить его на *произвольный* терм t и *всё ещё* получить кортеж термов из данного языка. Данная лемма формализует тот факт, что язык ограничений над теорией АТД может описывать только равенства и неравенства между подтермами ограниченной глубины: если пойти достаточно глубоко и заменить листовые термы произвольными термами, то начальный и результирующий термы будут *неотличимы* формулой языка первого порядка.

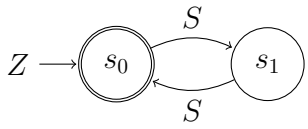
Теорема 19. Существуют регулярные, но неэлементарные инварианты, т. е. $\textsc{Reg} \setminus \textsc{Elem} \neq \varnothing$.

Доказательство. Рассмотрим систему дизъюнктов Хорна над алгебраическим типом целых чисел Пеано $Nat ::= Z \mid S\ Nat$, которая проверяет чётность чисел и утверждает, что никакие два соседних числа не могут быть чётными:

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

Система из этого примера имеет единственный индуктивный инвариант — множество $E = \{S^n(Z) \mid n \geqslant 0\}$. Это можно доказать от противного: если расширить это множество некоторым нечетным числом $E \cup \{S^{2n+1}(Z)\} \subseteq E'$, то будет нарушено условие запроса при $x = S^{2n}(Z)$ и $y = S^{2n+1}(Z)$. Таким образом, множество E оказывается единственным безопасным индуктивным инвариантом этой системы.

Легко видеть, что множество E выразимо следующим автоматом над деревьями (и следовательно, система имеет индуктивный инвариант в классе $\textsc{Reg}$):



Докажем, что множество E невыразимо формулой языка ограничений. Предположим, что множество E элементарно. Возьмём постоянную $K > 0$ из леммы 8. Пусть $g \equiv S^{2K}(Z) \in E, \sigma = Nat, p = S^{2K}$. Далее, $\bigcup_j leaves_\sigma(g_j) = leaves_\sigma(g) = \{p\}$, поэтому $P = \{p\}$. Тогда по лемме 8 найдётся такое $N \geqslant 0$, что если взять $t \equiv S^{2N+1}(Z)$, то $g[P \leftarrow t] \equiv S^{2K}(S^{2N+1}(Z)) \in E$. Следовательно, множеству E принадлежит нечётное число, что противоречит определению этого множества чётных чисел. □

Сформулируем аналогичную лемму для языка первого порядка с ограничениями на размер терма. Для этого рассмотрим соответствующее расширение селекторами SizeElem*, которое допускает устранение кванторов.

Теорема 20 (см. [106]). Всякая SizeElem-формула эквивалентна некоторой бесквантворной SizeElem*-формуле.

Определение 30. Заимствуя обозначения из работы [102], положим $\mathbb{T}^k_\sigma = \{t \text{ имеет сорт } \sigma \mid size(t) = k\}$. Для каждого σ, являющегося АТД-сортом, определим множество размеров термов $\mathbb{S}_\sigma = \{size(t) \mid t \in |\mathcal{H}|_\sigma\}$. *Линейное множество* — это множество вида $\{\mathbf{v} + \sum_{i=1}^n k_i \mathbf{v_i} \mid k_i \in \mathbb{N}_0\}$, где все $\mathbf{v}$ и $\mathbf{v_i}$ являются векторами над $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$.

Определение 31. АТД-сорт σ называется *расширяющимся*, если для каждого натурального числа n существует граница $b(\sigma, n) \geqslant 0$ такая, что для каждого $b' \geqslant b(\sigma, n)$, если $\mathbb{T}^{b'}_\sigma \neq \varnothing$, то $\left|\mathbb{T}^{b'}_\sigma\right| \geqslant n$. Сигнатура АТД называется расширяющейся, если все её сорта расширяющиеся.

Лемма 9 (Лемма о «накачке» для класса SizeElem). Пусть есть некоторая расширяющая АТД-сигнатура и $\mathbf{L}$ — элементарный язык n-кортежей с ограничениями на размер теремов. Тогда справедливо, что существует константа $K > 0$ такая, что:

- для каждого кортежа термов $\langle g_1, \ldots, g_n \rangle \in \mathbf{L}$,
- для любого i такого, что $\mathcal{H}eight(g_i) > K$,
- для всех бесконечных сортов $\sigma \in \Sigma_S$ и
- для всех путей p в терме g с глубиной большей K

– существует бесконечное линейное множество $T \subseteq \mathbb{S}_\sigma$ такое, что
– для любых термов t сорта σ с размером $size(t) \in T$,
– существует набор последовательностей путей P_j, в котором ни один путь не является суффиксом пути p,
– и набор последовательностей термов U_j такие, что

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

Доказательство. Доказательство приведено в работе [37]. □

Основная идея леммы 9 заключается в том, что имея язык из класса SizeElem, достаточно большой терм g из него и достаточно большой путь p, можно заменить $p\,(g)$ произвольным термом t (его размер должен находиться в некотором линейном бесконечном множестве T), и вновь получить терм из этого языка. Данный факт, в свою очередь, означает, что в каждом бесконечном языке из класса SizeElem существуют подтермы, которые неотличимы его формулами.

Пример 15 (*even*). Рассмотрим систему дизъюнктов Хорна над алгебраическим типом бинарных деревьев $Tree ::= leaf \,|\, node(Tree, Tree)$, которая проверяет, является ли количество узлов в самой левой ветви дерева чётным.

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

Как показано ниже, у этой системы не существует инварианта, выразимого элементарно даже с ограничениями на размер термов.

Теорема 21. Существуют регулярные инварианты, которые не являются инвариантами, выразимыми элементарно с ограничениями на размер термов, т. е. $even \in \text{Reg} \setminus \text{SizeElem}$.

Доказательство. Инвариант системы дизъюнктов Хорна *even* выражает автомат $\langle \{s_0, s_1\}, \Sigma_F, \{s_0\}, \Delta \rangle$ с правилами перехода Δ, описываемыми следующим

образом:

$$\begin{aligned} leaf &\mapsto s_0 \\ node(s_0, s_0) &\mapsto s_1 \\ node(s_0, s_1) &\mapsto s_1 \\ node(s_1, s_0) &\mapsto s_0 \\ node(s_1, s_1) &\mapsto s_0 \end{aligned}$$

Используя лемму о «накачке», можно доказать, что инвариант *even* не лежит в классе SizeElem. Во-первых, очевидно, что сорт *Tree* является расширяющимся. Предположим, что *even* находится в классе SizeElem и имеет инвариант $\mathbf{L}$. Возьмём $K > 0$ из леммы 9. Пусть $g \in \mathbf{L}$ будет полным двоичным деревом высоты $2K$, $\sigma = Tree, p = Left^{2K}$. Возьмём бесконечное линейное множество T из леммы. Мы можем найти некоторое $n \in T, n > 2$ и $t = node(leaf, t')$ для некоторого t' такие, что $size(t) = n$. По лемме 9 существует последовательность путей P и последовательность термов U, и ни один из элементов в P не является суффиксом p; также должно быть верно, что $g[p \leftarrow t, P \leftarrow U] \in \mathbf{L}$, поэтому крайний левый путь в дереве должен иметь чётную длину. Однако по крайнему левому пути $p = Left^{2K}$ лежит терм $node(leaf, t')$, поэтому путь до крайнего левого листа дерева имеет длину $2K - 1 + 2 = 2 + 1$, являясь нечётным числом. Итак, имеем противоречие с тем, что путь до крайнего левого листа в каждом терме множества *even* имеет чётную длину, следовательно *even* не лежит в классе SizeElem. □

5.3 Конечные представления множеств термов

В данной диссертационной работе были рассмотрены и предложены различные классы инвариантов для систем дизъюнктов Хорна над АТД, такие, как Elem, Reg, Reg_+, $\text{Reg}_\times$ и т. д. Ключевое требование к классам индуктивных инвариантов над АТД — это возможность представлять бесконечные множества кортежей термов конечным образом, чтобы с ними мог работать конечный вычислитель. Кроме этого, от них требуется замкнутость и разрешимость некоторых операций, которые были подробно рассмотрены в этой главе. Конечные представления множеств термов с такими свойствами исследуются в других областях информатики и могут быть использованы для вывода инвариантов.

Одной из альтернативных формулировок задачи конечного представления множеств термов является задача представления эрбрановских моделей, которой занимается область автоматического построения моделей (automated model building) [107]. Основная задача этой области — автоматически построить модель формулы логики первого порядка, когда её опровержение не может быть найдено. По теореме Эрбрана, формула выполнима тогда и только тогда, когда у неё есть эрбрановская модель, поэтому достаточно строить только эрбрановские модели, которые в общем случае содержат в себе бесконечные множества термов. Для автоматизации построения таких моделей рассматривают различные конечные их представления [108—111]. В частности, в этих работах приведены эффективные алгоритмы для работы с моделями, представленными автоматами над деревьями и их расширениями. Качественный обзор вычислительных представлений эрбрановских моделей, их свойств, выразительной силы и эффективности необходимых для работы с ними процедур представлен в работах [112; 113]. Хотя предложенные в данных работах представления могут быть использованы для представления инвариантов над АТД, создание алгоритмов вывода таких инвариантов остаётся трудоёмкой задачей, которая не затрагивалась в этих работах.

Задачу конечного представления множеств термов можно также сформулировать в контексте формальных языков деревьев как задачу построения расширений автоматов над деревьями, обладающих свойствами разрешимости и замкнутости базовых языковых операций, рассмотренных в этой главе. Языки деревьев систематически рассматриваются в контексте формальных языков [114], в частности, существует множество работ, предлагающих внедрение различных видов синхронизации в автоматы над деревьями [81—86]. Однако с предлагаемыми в этой области представлениями есть несколько ограничений. С одной стороны, чаще всего предлагаются языки с эффективным (полиномиальным с низкой степенью полинома) алгоритмом парсинга (принадлежности кортежа термов языку), из-за чего предлагаемые языки имеют низкую выразительную силу. С другой стороны, часто предлагаются классы языков деревьев, не замкнутые относительно некоторых булевых операций, например, отрицания и пересечения, что делает задачу адаптации этих классов для вывода индуктивных инвариантов ещё более трудоёмкой.

Стоит отдельно упомянуть работы по расширению автоматов над деревьями SMT-ограничениями из других теорий до т. н. символьных автоматов

над деревьями [115; 116]. Класс инвариантов, построенный на таких автоматах, позволит проверять выполнимость систем дизъюнктов Хорна над комбинацией АТД с другими SMT-теориями, как было замечено в работе [117]. Авторы этой работы начали адаптацию символьных автоматов к задаче проверке выполнимости систем дизъюнктов Хорна в рамках, реализовав учитель для этого класса инвариантов в рамках подхода ICE. Дальнейшее исследование класса инвариантов, построенного на символьных автоматах над деревьями, в рамках задачи автоматического вывода инвариантов представляется наиболее перспективным.

Итак, конечные представления множеств кортежей термов, представленные в работах из этих областей, могут стать основой для будущих классов индуктивных инвариантов над АТД. Поскольку многие из них построены как расширения классов, рассмотренных в данной работе, предложенные в данной работе методы вывода инвариантов могут быть также адаптированы, чтобы выводить инварианты в новых классах.

5.4 Выводы

Среди всех классов инвариантов программ, для которых существуют эффективные процедуры автоматического вывода инвариантов, наиболее выразительными являются классы $\text{Reg}_\times$ и ElemReg. Они позволяют как выражать сложные рекурсивные отношения, так и синхронные отношения, и следовательно, они расширяют применимость автоматического вывода инвариантов на практике. Однако из-за высокой выразительной силы автоматическое построение инвариантов в этих классах может быть затруднено в виду роста сложности примитивных операций. В следующей главе приведено сравнение эффективности существующих и предложенных методов вывода инвариантов для рассмотренных классов.

Глава 6. Реализация, сравнения и эксперименты

6.1 Пилотная программная реализация

Все предложенные в данной работе подходы реализованы в созданном в рамках данной работы Хорн-решателе RInGen (*R*egular *In*variant *Gen*erator)[1]. Реализация занимает 5200 строк на языке F#. Было принято решение реализовать Хорн-решатель с нуля, не встраиваясь в кодовые базы существующих решателей, поскольку предложенные алгоритмы требуют нетривиальных манипуляций с формулами и результатами других логических решателей.

Общая архитектура реализованного инструмента представлена на рисунке 6.1. Инструмент RInGen принимает на вход систему дизъюнктов Хорна с ограничениями в формате SMTLIB2 [118]. После парсинга дизъюнктов выполняется их упрощение, в частности, устранение равенств, селекторов и тестеров. Затем, в зависимости от поданных Хорн-решателю опций, запускается один из алгоритмов, предложенных в данной работе. Так, «Модуль замены АТД неинтерпретированными функциями» реализует алгоритм из главы 2, а «Модуль создания декларативного описания автомата над деревьями» — алгоритм из главы 3. Результатом работы каждого из них является формула над неинтерпретированными функциями, которая передаётся в сторонний логический решатель — инструмент автоматического доказательства теорем Vampire или SMT-решатель cvc5. В итоге инструмент возвращает безопасный индуктивный инвариант, если система безопасна, в противном случае — резолютивное опровержение.

RInGen. Подход, представленный в главе 2, реализован автором данного исследования в рамках Хорн-решателя RInGen. Как сам подход, так и его реализация подразумевают использование стороннего SMT-решателя $\mathcal{V}$ для теории неинтерпретированных функций с кванторами, поэтому предлагаемая реализация в дальнейшем будет обозначаться RInGen($\mathcal{V}$). В частности, в качестве решателя $\mathcal{V}$ в экспериментах используются инструмент Vampire [76] и SMT-решатель cvc5. Инструмент Vampire использует портфолио-подход [119], то есть перебирает различные техники проверки выполнимости формул, построенные на насыщении системы [96] или на поиске конечных моделей [92].

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

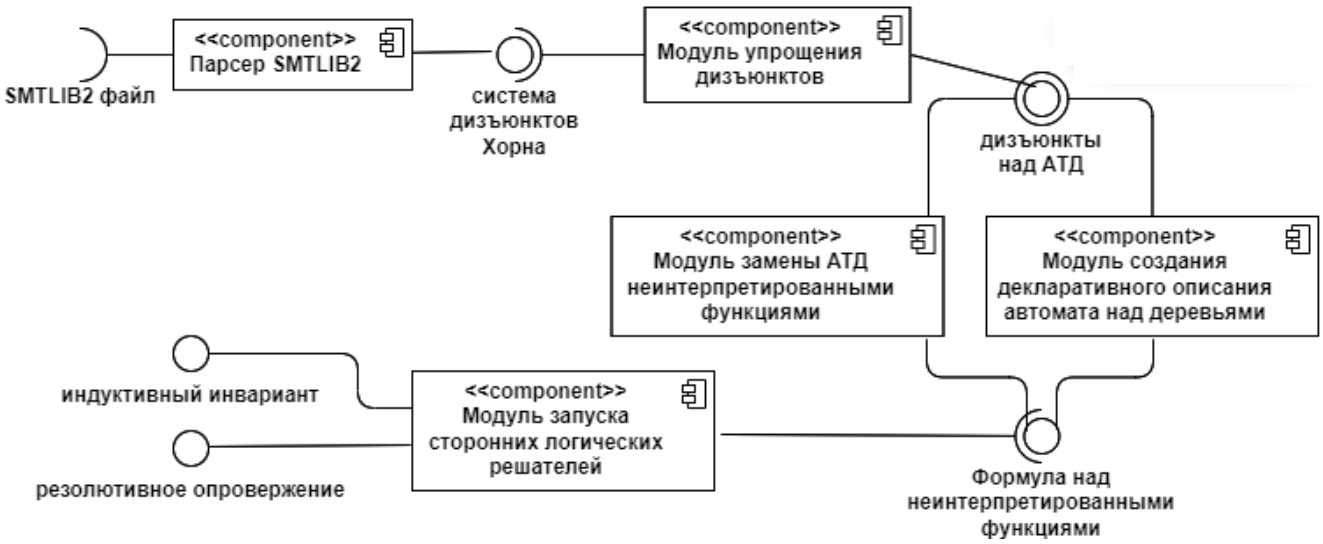


Рисунок 6.1 — Архитектура Хорн-решателя RInGen

Инструмент CVC5 используется в режиме построения конечных моделей[2] [91]. Оба инструмента позволяют как доказывать безопасность системы, так и находить контрпримеры.

RInGen-Sync. Этот подход, представленный в главе 3, реализован как надстройка над RInGen($\mathcal{V}$). В качестве стороннего решателя $\mathcal{V}$ в экспериментах использовался CVC5, поскольку RInGen-Sync порождает символы с большой арностью, которые не поддерживаются инструментом Vampire[3]. Эта реализация в дальнейшем будет обозначаться RInGen-Sync[4].

RInGen-CICI. Данный подход, представленный в главе 4, реализован в рамках кодовой базы Хорн-решателей Racer [25] (развитие Хорн-решателя Spacer [24], реализованное в логическом решателе Z3[5]) и Хорн-решателя RInGen($\mathcal{V}$)[6], описанного выше. Эта реализация в дальнейшем будет обозначаться RInGen-CICI($\mathcal{V}$). Далее описаны обе части этой реализации в Хорн-решателях Racer и RInGen($\mathcal{V}$), соответственно, которые далее называются *базовыми* относительно инструмента RInGen-CICI($\mathcal{V}$).

Хорн-решатель Racer разработан Ари Гурфинкелем (Arie Gurfinkel) и Хари Говинд Ведирамана Кришнаном (Hari Govind Vediramana Krishnan) из университета Ватерлоо. Инструмент Racer основан на подходе, называемом достижимость, направляемая свойством (Property-Directed Reachability,

[2] с опцией `--finite-model-find`

[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

PDR) [24], который можно рассматривать как сложный экземпляр CEGAR. PDR строит абстрактные состояния в виде конъюнкции формул (называемых *леммами*) на различных *уровнях* путём итеративного увеличения уровня в цикле. При этом поддерживаются следующие возможности: если набор лемм $\{\varphi_i\}$ был построен на уровне n, то $\bigwedge_i \varphi_i$ аппроксимирует сверху все состояния, достижимые менее чем за n шагов перехода, и аппроксимирует снизу свойство безопасности. Таким образом, леммы в PDR выполняют требование абстракции в процедуре CollaborateCHCs (листинг 4.4). Хорн-решатель Racer был модифицирован в рамках данного диссертационного исследования таким образом, чтобы в конце каждой итерации набор лемм последнего уровня асинхронно передавался новому процессу инструмента RInGen($\mathcal{V}$).

Процедура CollaborateCHCs (листинг 4.4) реализована на основе инструмента RInGen($\mathcal{V}$). Было выполнено следующее обобщение в процедуре ResidualCHCs$(\mathcal{P}, a)$ (см. 4.2.3). Конъюнктивная форма лемм инструмента Racer используется для вывода инвариантов более общего вида: $\bigwedge_i(\varphi_i(\overline{x}) \vee \overline{x} \in L_i)$. Так, имея $a(P) = \bigwedge_i \varphi_i$, мы заменяем все атомы $P(\overline{t})$ на *конъюнкцию дизъюнкций* $\bigwedge_i(\varphi_i(\overline{t}) \vee L_i(\overline{t}))$ с новыми предикатными символами L_i. Это позволяет выводить более общие инварианты, чем инварианты из объединения классов Elem и $\mathcal{A}$ (см. определение 25), которое состоит только из формул вида $\varphi(\overline{x}) \vee \overline{x} \in L$.

После преобразований модифицированный RInGen($\mathcal{V}$) вызывает сторонний решатель $\mathcal{V}$ с ограничением по времени в 30 секунд. Затем его результаты передаются обратно в Хорн-решатель Racer, где они асинхронно обрабатываются. Кроме того, реализация не выполняет дорогостоящее преобразование в КНФ из листинга 4.5, так как реализация RInGen($\mathcal{V}$) позволяет принимать на вход дизъюнкты Хорна в произвольном виде, поскольку он полагается на сторонний решатель $\mathcal{V}$ с полной поддержкой логики первого порядка.

6.2 Сравнение и соотнесения

Данный раздел посвящен сравнению предложенных методов решения систем дизъюнктов Хорна с алгебраическими типами данных и существующих методов, реализованных в таких инструментах, как Spacer, Racer, Eldarica, VeriCaT, HoIce и RCHC. Данные инструменты были отобраны по следующему принципу: инструменты, поддерживающие системы дизъюнктов

Хорна над алгебраическими типами данных, которые проверяют как выполнимость, так и невыполнимость этих систем. Так, например, не рассматривались инструменты, решающие родственную проблему автоматизации индукции для теорем с алгебраическими типами данных, такие как, например, CVC5 в режиме индукции [120], AdtInd [121] и пр., поскольку они не принимают на вход системы дизъюнктов Хорна. Также не рассматривались инструменты логического программирование (такие, как Prolog [122]), поскольку они позволяют проверять только невыполнимость систем дизъюнктов Хорна и ничего не говорят об их выполнимости.

Таблица 6.1 — Сравнение Хорн-решателей с поддержкой АТД

Инструмент	Класс инвариантов	Метод	Возвращает инвариант	Полностью автоматический
Spacer	Elem	IC3/PDR	Да	Да
Racer	CatElem	IC3/PDR	Нет	Нет
Eldarica	SizeElem	CEGAR	Да	Да
VeriCaT	–	Трансф.	Нет	Да
HoIce	Elem	ICE	Да	Да
RCHC	Reg_{+}	ICE	Да	Да
RInGen(cvc5)	Reg	Трансф. + FMF	Да	Да
RInGen(Vampire)	–	Трансф. + Насыщение	Нет	Да
RInGen-Sync	$\text{Reg}_{\times}$	Трансф. + FMF	Да	Да
RInGen-CICI(cvc5)	ElemReg	CEGAR($\mathcal{O}$)	Да	Да
RInGen-CICI(Vampire)	–	CEGAR($\mathcal{O}$)	Нет	Да

В таблице 6.1 представлены результаты сравнения работе Хорн-решателей — существующих (верхний блок) и предложенных в данной (нижний блок). Предложенные Хорн-решатели описаны в предыдущей главе 6, реализуемые ими методы описаны в главах 2, 3 и 4 данной работы. В таблице, для краткости, название RInGen сокращено до RInGen, так, например, Хорн-решатель RInGen-Sync в таблице представлен как RInGen-Sync. Под словом «Трансф.» имеется в виду, что инструмент построен на применении нетривиальных трансформаций к системе; аббревиатура «FMF» обозначает применение

автоматического поиска конечных моделей («finite-model finding», см., например, [91; 92]); прочерк в столбце «Класс инвариантов» означает следующее: несмотря на то, что при выполнимости системы вывод инструмента неявно кодирует её индуктивный инвариант, не существует всюду останавливающейся процедуры, позволяющей этот вывод проверить. Остальные обозначения поясняются в подразделах, посвящённых соответствующим инструментам.

Большинство рассмотренных инструментов отличаются классами, в которых они ищут индуктивные инварианты. Сравнение самих классов инвариантов было приведено в главе 5. В контексте сопоставления инструментов сравнение их классов инвариантов важно по следующей причине: если инструмент выводит инварианты в некотором классе, то проблема невыразительности этого класса (невозможность выразить определённые типы отношений) превращается в проблему незавершаемости этого инструмента. Иными словами, поскольку ни один из существующих инструментов не проверяет, существует ли *вообще* инвариант для данной системы дизъюнктов Хорна в его классе[7], то в случае отсутствия такового инструмент не будет завершаться.

Далее приведено краткое сравнительное описание существующих инструментов.

Инструмент Spacer [24] строит элементарные модели (класс Elem). Этот инструмент создан на основе классической разрешающей процедуры для АТД, а также процедуры интерполяции и устранения кванторов [125]. Ядром инструмента является подход Spacer, который основан на технике, называемой *достижимость, направляемая свойством* (property-directed reachability, IC3/PDR), которая равномерно распределяет время анализа между поиском контрпримеров и построением безопасного индуктивного инварианта, распространяя информацию о достижимости небезопасных свойств и частичные леммы о безопасности. Инструмент позволяет выводить инварианты в комбинации алгебраических и других типов данных, возвращает проверяемые сертификаты. Подход, используемый в инструменте, корректен и полон. Недостатком инструмента является то, что он выражает инварианты в языке ограничений, а потому часто не завершается на проблемах с АТД.

[7] С одной стороны, эта задача по сложности сравнима с самой задачей верификации, а с другой стороны до сих пор ей были посвящены лишь отдельные работы (см., например, [123; 124])

Инструмент Racer [25] является развитием инструмента Spacer, позволяя выводить инварианты в языке ограничений, расширенном катаморфизмами. Этот язык ограничений обозначен в таблице 6.1 как CatElem. Racer также наследует все достоинства подхода Spacer. Недостатком подхода является то, что он не полностью автоматический, поскольку требует вручную описывать катаморфизмы, что может быть затруднительно на практике, поскольку по заданной проблеме бывает сложно понять, какие катаморфизмы потребуются для её инварианта. Недостатком самого инструмента является то, что он не возвращает какие-либо проверяемые сертификаты с катаморфизмами.

Инструмент Eldarica [26] строит модели с ограничениями размера термов, которые вычисляют общее количество вхождений конструкторов (класс SizeElem). Это расширение весьма ограниченно увеличивает выразительность языка ограничений, поскольку введённая функция считает количество всех конструкторов одновременно, поэтому с её помощью невозможно выразить многие свойства, например, ограничение на высоту дерева. Инструмент Eldarica использует подход CEGAR с абстракцией предикатов и встроенный SMT-решатель Princess [75], который предоставляет разрешающую процедуру, а также процедуру интерполяции для АТД с ограничениями на размер термов. Эти процедуры построены на сведении данной теории к комбинации теорий неинтерпретируемых функций и линейной арифметики [102].

Инструмент VeriCaT [31—34] обрабатывает условия проверки над теориями линейной арифметики и АТД и полностью устраняет АТД из исходной системы дизъюнктов путём сворачивания (fold), разворачивания (unfold), введения новых дизъюнктов и других трансформаций. После работы инструмента получается система дизъюнктов Хорна без АТД, на которой может быть запущен любой эффективный Хорн-решатель, например, Spacer или Eldarica. Основным достоинством подхода является тот факт, что он рассчитан на работу с проблемами, где алгебраические типы данных комбинированы с другими теориями. Основные недостатки подхода заключаются в следующем: сам процесс трансформации может также не завершаться, кроме этого из-за трансформации невозможно восстановить инвариант исходной системы, т. е. инструмент не возвращает проверяемого сертификата.

Инструмент HoIce [27] строит элементарные инварианты с помощью подхода, основанного на машинном обучении, ICE [54]. Его достоинством является возможность выводить инварианты в комбинации АТД с другими теориями, а также корректность и полнота, и, наконец, способность возвращать проверяемые сертификаты корректности. Его недостатком является то, что он выводит инварианты в невыразительном языке ограничений, а потому часто не завершается.

Инструмент RCHC [28; 126] также использует подход ICE; RCHC основан на машинном обучении, однако выражает индуктивные инварианты программ над АТД при помощи *автоматов над деревьями* [35]. Но из-за сложностей с выражением кортежей термов автоматами, описанных в разделе 5.2.1, подход часто оказывается неприменим для простейших примеров, где существуют классические символьные инварианты.

Выводы. Подводя итоги сравнения вышеозначенных инструментов и методов, можно сказать следующее. По сравнении с методом инструмента RCHC предложенные в данной диссертационной работе подходы являются альтернативными способами вывода регулярных инвариантов и их надклассов. Поэтому они могут быть совмещены с подходом инструмента RCHC, чтобы быстрее сходиться к индуктивному инварианту системы, если он существует. В сравнении с методами остальных существующих инструментов, предложенные подходы позволяют выводить инварианты в независимых классах регулярных инвариантов. Поэтому применение предложенных методов совместно с существующими позволит решать больше различных типов задач.

6.3 Дизайн эксперимента

6.3.1 Выбор инструментов для сравнения

В качестве инструментов для сравнения были выбраны Racer [25] и Eldarica [26] — Хорн-решатели с поддержкой алгебраических типов данных, лидирующие на соревнованиях CHC-COMP [30]. Также были выбраны инструменты cvc5-Ind (cvc5 в режиме индукции) [120] и VeriCaT [31]. Несмотря на то, что эти инструменты не строят индуктивные инварианты *явно*, из-за чего невозможно проверить их корректность, их запуск на эквивалентном бенчмарке

добавлен в экспериментальное сравнение, поскольку они решают родственную задачу.

В представленных здесь экспериментах не участвовал Хорн-решатель HoIce [27], поскольку он работает не быстрее Racer [25] и при этом ищет инварианты в том же классе инвариантов Elem. Другой известный Хорн-решатель RCHC [28] не участвовал в экспериментах, поскольку он работает нестабильно и часто либо завершается с ошибкой, либо возвращает некорректные результаты.

6.3.2 Тестовый набор данных

Эксперименты проводились на наборе данных *TIP* (Tons of Inductive Problems) [127], тестового набора с трека соревнования CHC-COMP 2022[8], посвящённого алгебраическим типам данных. Набор *TIP* состоит из 454 систем дизъюнктов Хорна, полученных из Haskell-программ с АТД и рекурсией. В тестовом наборе встречаются следующие алгебраические типы данных: списки, очереди, регулярные выражения и целые числа Пеано.

6.3.3 Описание тестового стенда

Эксперименты проводились на платформе StarExec [128], имеющей кластер машин Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz и Red Hat Enterprise Linux 7[9], с ограничением на процессорное время работы каждого инструмента на одном тесте в 600 секунд и с ограничением по памяти в 16 ГБ.

6.3.4 Исследовательские вопросы

Для постановки экспериментов были поставлены следующие исследовательские вопросы.

Исследовательский вопрос 1 (Количество решений). Поскольку основная цель данной работы — предложить подходы, позволяющие проверять выполнимость б*о*льшего числа систем, чем аналоги, путём вывода индуктивных инвариантов, ключевыми являются следующие вопросы.

– Позволяют ли предложенные подходы проверять выполнимость б*о*льшего числа систем, чем подходы, строящие классические символьные инварианты?

[8] https://github.com/chc-comp/ringen-adt-benchmarks

[9] https://www.starexec.org/starexec/public/machine-specs.txt

– Позволяют ли предложенные подходы проверять выполнимость систем, у которых существуют классические инварианты?

Исследовательский вопрос 2 (Производительность).

– Какова производительность инструментов RInGen и RInGen-Sync на проблемах, которые смогли решить они, а также существующие инструменты?
– Коллаборативный вывод в RInGen-CICI может потребовать параллельного запуска нескольких экземпляров оракула. Каково влияние параллельного запуска на производительность?

Исследовательский вопрос 3 (Значимость класса индуктивных инвариантов). Коллаборативный вывод, реализованный в RInGen-CICI, теоретически позволяет не только выводить инварианты в большем классе, но и ускорять сходимость поиска классических символьных инвариантов. Какова при этом доля классических символьных инвариантов на всех уникально решённых инструментом RInGen-CICI проблемах?

6.4 Анализ результатов экспериментов

6.4.1 Количество решений

Количество проблем из тестового набора, решённых существующими и предложенными инструментами, представлено в таблице 6.2. Над разделяющей чертой находятся существующие инструменты, а предложенные инструменты расположены под ней.

RInGen. На всех 12 проблемах, на которых инструмент Eldarica вернул ответ «UNSAT», инструмент RInGen завершился с тем же результатом и он нашёл больше контрпримеров. Инструменты RInGen(cvc5), RInGen(Vampire) и Racer нашли контрпримеры для 21, 46 и 22 систем дизъюнктов соответственно, при этом каждый из них нашёл несколько уникальных контрпримеров. Инструмент RInGen(Vampire) построил существенно больше «UNSAT» результатов, чем другие инструменты, поскольку в нём реализована эффективная процедура вывода опровержений. Тем самым, несмотря на то, что предложенные алгоритмы спроектированы для поиска большего числа индуктивных инвариантов, они также позволяют находить уникальные контрпримеры. Далее, инструмент Eldarica нашёл 46 инвариантов в противовес

Таблица 6.2 — Результаты экспериментов. «SAT» обозначает, что система безопасна (есть индуктивный инвариант), «UNSAT» обозначает, что система небезопасна.

Инструмент	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

25 и 135 инвариантам, найденным RInGen(cvc5) и RInGen(Vampire). Из них Eldarica решила 25 уникальных (не решённых RInGen(cvc5)) задач, каждая из которых является формулировкой некоторого свойства порядковых предикатов ($<, \leqslant$, $>, \geqslant$) на числах Пеано. Эти проблемы легко решаются инструментом Eldarica, поскольку порядковые предикаты сами по себе включены в домен верификации SizeElem на уровне примитивов. Тем не менее инструмент RInGen(cvc5) решил 13 уникальных (не решённых Eldarica) проблем, чьи инварианты не представимы в домене верификации инструмента Eldarica. Эффективность реализованного в инструменте RInGen подхода существенно зависит от используемого стороннего решателя, как видно из того, что инструмент RInGen(Vampire) вывел более чем в 5 раз больше инвариантов, чем инструмент RInGen(cvc5).

RInGen-Sync. Данный инструмент завершился с ответом UNSAT на 21 проблеме, эти ответы в точности совпадают с 21 результатом инструмента RInGen(cvc5), поскольку поиск контрпримеров инструмент RInGen-Sync наследует от последнего.

Среди всех полученных инструментом Eldarica ответов SAT 38 были также получены инструментом RInGen-Sync. Большое количество пересечений с результатами инструмента Eldarica связано с тем, что Eldarica хорошо справляется с проблемами, которые кодируют порядок на числах Пеано, который также хорошо кодируется полносвёрточными синхронными автоматами над деревьями, используемыми в RInGen-Sync. Racer завершился с

ответом SAT на 26 системах, 15 из которых пересекаются с ответами инструмента RInGen-Sync. Также инструмент RInGen-Sync вывел 4 уникальных инварианта. Несмотря на то, что теоретически выразительная сила полносвёрточных синхронных автоматов над деревьями, используемых в RInGen-Sync, должна давать большее число решений, инструменты поиска конечных моделей, которые RInGen-Sync использует в качестве бэкенда, не завершаются на проблемах с большим количеством кванторов и потому RInGen-Sync часто не завершается. Результаты не меняются при изменении бэкенда с cvc5 на другие инструменты поиска конечных моделей и увеличении ограничения на время до 1200 секунд. Небольшое количество пересечений с инструментом Racer говорит о том, что хотя в теории класс элементарных инвариантов почти полностью содержится в классе синхронных регулярных инвариантов, на практике предложенный подход не позволяет эффективно выводить инварианты систем, у которых существуют элементарные инварианты.

RInGen-CICI решил меньше *небезопасных проблем*, чем лучший из базовых решателей: RInGen-CICI получил 19 (с cvc5) и 28 (с Vampire) UNSAT результатов против 21 (с cvc5) и 46 (с Vampire) UNSAT результатов, полученных RInGen. Основная причина в том, что предложенный подход спроектирован для решения более сложной задачи вывода индуктивных инвариантов и не вносит изменения в работу базовых алгоритмов поиска контрпримеров. То есть, с нашим подходом могут быть интегрированы ортогональные усовершенствования поиска контрпримеров, например, предложенные в [129]. Таким образом, все контрпримеры, полученные RInGen-CICI, получены непосредственно от одного из базовых решателей. Некоторые контрпримеры, которые находит инструмент RInGen, не были найдены RInGen-CICI, поскольку он запускает RInGen с ограничением по времени в 30 секунд.

Важно отметить, что все 20 SAT и 15 UNSAT ответов, полученных Racer, также были получены и инструментом RInGen-CICI, за исключением одного UNSAT ответа.

На безопасных проблемах инструмент RInGen-CICI превзошёл конкурирующие решатели: RInGen-CICI(cvc5) получил 117 SAT ответов, когда как Racer получил 20 SAT ответов, а RInGen(cvc5) — 25, также RInGen-CICI(Vampire) получил 189 SAT ответов, при 20 SAT ответах от Racer и 135 от RInGen(Vampire). Таким образом, RInGen-CICI решает значительно больше SAT-задач, чем базовые инструменты, работающие по

отдельности: 117 против 20 + 25 и 189 против 20 + 135 для соответствующих бэкендов CVC5 и VAMPIRE. В частности, RINGEN-CICI(CVC5) решает 97 задач, не решённых RACER и 94 задачи, не решённые RINGEN(CVC5). RINGEN-CICI(VAMPIRE) решает 169 задач, не решённых RACER, и 60 задач, не решённых RINGEN(VAMPIRE). Таким образом, коллаборативный метод вывода инвариантов позволяет установить выполнимость существенно больше числа систем, чем параллельный запуск базовых инструментов.

Однако есть проблемы, которые были решены базовыми решателями, но не решены предложенным инструментом. Инструмент RINGEN-CICI(CVC5) не решил 7 проблем, которые были успешно решены инструментом RINGEN(CVC5). Две из этих проблем могут быть решены предложенным инструментом, если увеличить 30-секундное ограничение по времени для работы бэкенда в RINGEN-CICI. Существующие методы предсказания времени проверки, такие как [130], могут быть применены для того, чтобы вообще избежать жесткого кодирования лимита времени. Остальные 5 проблем решаются мгновенно, однако их результаты не могут быть получены из межпроцессного взаимодействия в сделанной реализации. Причина заключается в том, что RACER тратит слишком большое время на решение SMT-ограничений, и поэтому не считывает результаты бэкенд-решателя. Этой технической проблемы можно избежать, если считывать результаты бэкенд-решателя в более частых контрольных точках, что, однако, приведёт к увеличению накладных расходов в инструменте. Аналогичная картина наблюдается и для RINGEN-CICI(VAMPIRE), который не смог решить 24 проблемы, решённые базовыми решателями. Только 8 из них не решены из-за низкого ограничения по времени для бэкенда, а остальные 16 — из-за расхождения RACER в решении SMT-ограничений.

Итак, инструмент RINGEN(CVC5) не опередил существующие решения, однако дал множество уникальных решений по сравнению с ними, поскольку выводил инварианты в новом классе. Инструмент RINGEN(VAMPIRE), основанный на том же подходе, решил более чем в 2.5 раза больше проблем, чем лучший из существующих инструментов, по той же причине. Несмотря на то, что класс инвариантов инструмента RINGEN-SYNC существенно шире, вывод инвариантов в нём гораздо более трудоёмкий, поэтому хотя он смог вывести почти в два раза больше инвариантов, чем RINGEN(CVC5), он не превзошёл лучший из существующих инструментов. Лучшие результаты показал инструмент

RInGen-CICI(cvc5), который решил на 235% больше задач, чем параллельная композиция Racer и RInGen(cvc5), а также на 39% больше задач с бэкендом Vampire, благодаря балансу между размером класса инвариантов и эффективностью процедуры вывода инвариантов. Наилучший из предложенных инструментов RInGen-CICI(Vampire) всего решил 189 + 28 проблем, что примерно в 3.74 раза больше, чем наилучший из существующих инструментов Eldarica, решивший 46 + 12 проблем.

6.4.2 Производительность

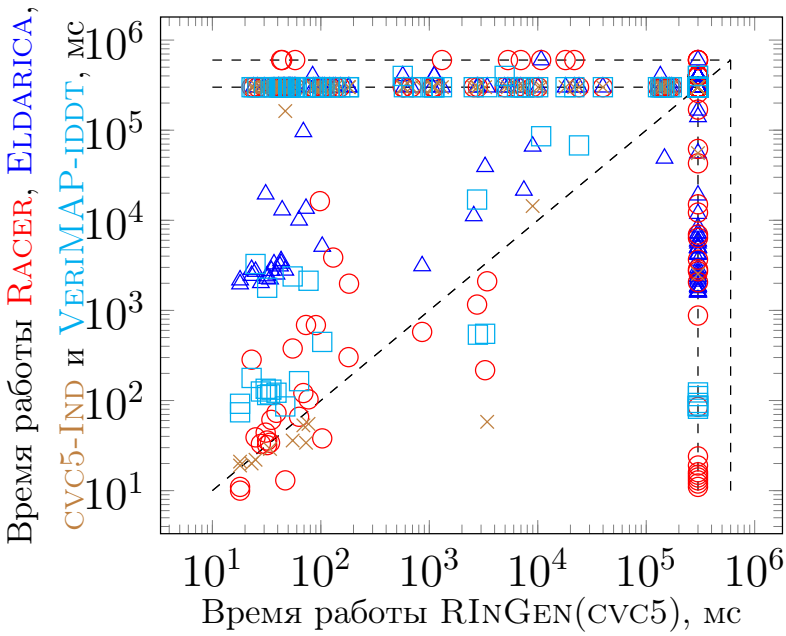


Рисунок 6.2 — Сравнение производительности инструментов. Каждая точка на графике представляет пару длительностей выполнения.

Графики на рисунке 6.2 показывают, что инструмент RInGen(cvc5) не только выводит больше инвариантов, но и работает быстрее, чем другие инструменты, в среднем, на один порядок. На рисунке некоторые небезопасные системы проверялись быстрее инструментами cvc5-Ind, VeriCaT и Racer. Это может быть связано с более эффективной процедурой инстанцирования кванторов в cvc5-Ind и более сбалансированным компромиссом между выводом инвариантов и поиском контрпримеров в ядре Racer (который также вызывается инструментом VeriCaT). На проблемах, решённых несколькими

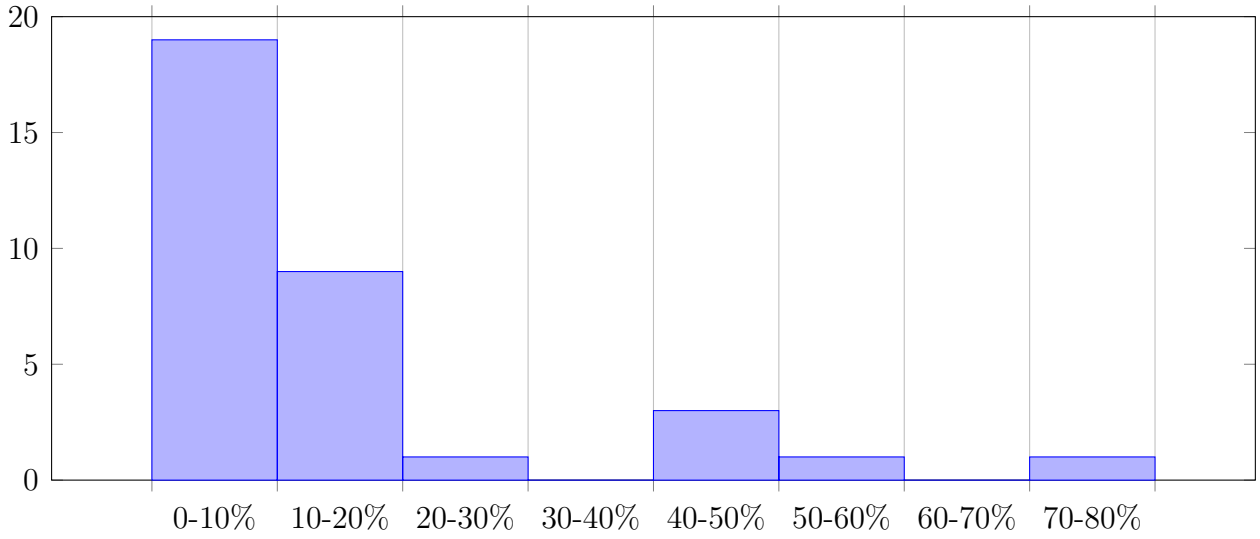


Рисунок 6.3 — Количество тестовых примеров (ось y), решённых как RInGen-CICI, так и Racer, и затраты процессорного времени (ось абсцисс) на выполнение RInGen-CICI по сравнению с Racer. Racer превзошел RInGen-CICI на 34 запусках. Нет ни одного запуска с накладными расходами более 80%, поэтому далее ось абсцисс не показана.

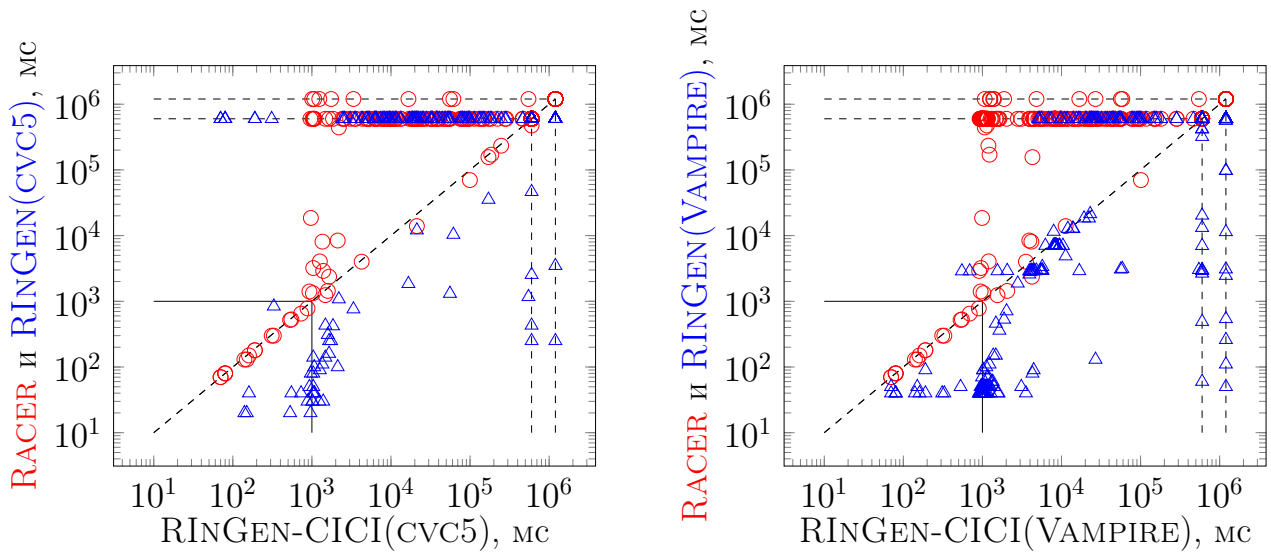


Рисунок 6.4 — Сравнение времени работы инструментов

инструментами, инструмент RInGen(cvc5) работает в среднем на два порядка быстрее. Запуски RInGen(Vampire) заняли ещё меньше времени, чем RInGen(cvc5).

RInGen-Sync и RInGen-CICI. На рисунке 6.4 представлены *общие графики производительности* для инструмента RInGen-CICI по сравнению с базовыми решателями. Каждая точка на графике представляет собой время ра-

боты (в миллисекундах) RInGen-CICI (ось x) и конкурирующего инструмента (ось y): треугольниками обозначен RInGen, а кругами — Racer. На внешних пунктирных линиях представлены ошибки инструментов, которые имели место как для Racer, так и для RInGen-CICI, из-за нестабильности используемой версии Racer[10]. Внутренние пунктирные линии обозначают случаи, когда решатель достиг ограничения по времени. Поскольку RInGen-CICI решил значительно больше экземпляров, чем конкурирующие решатели, большая часть фигур находится на верхних пунктирных линиях обоих графиков. Половина оставшихся фигур находится около диагонали, что означает, что совместная работа завершилась после первого совместного вызова решателя. Другая половина фигур находится около одной секунды (которая отмечена сплошной линией в левом нижнем углу) по той же причине, по которой некоторые проблемы не решены: внутренний движок Racer в RInGen-CICI выполняет решение сложных SMT-ограничений и поэтому не считывает результат бэкенда в течение некоторого времени. Большинство кругов, не попавших на пунктирные линии, находятся около диагонали на обоих графиках, и это означает, что RInGen-CICI работал сопоставимо с Racer на проблемах, решённых обоими инструментами.

На рисунке 6.3 представлен график, демонстрирующий *накладные расходы* коллаборативного вывода в RInGen-CICI. Имеется всего 34 и 35 проблем, решённых одновременно Racer и RInGen-CICI(cvc5) и Racer и RInGen-CICI(Vampire), соответственно. В 35 из этих 69 запусков RInGen-CICI был быстрее инструмента Racer, а в остальных 34 — нет, потому что ни один вызов бэкенда не был успешным, а поэтому RInGen-CICI вёл себя так же, как Racer, но при этом имел накладные расходы на создание процессов. Накладные расходы на этих 34 запусках представлены на рисунке 6.3. На графике показано, во сколько раз медленнее работает RInGen-CICI по сравнению с Racer. Накладные расходы в большинстве запусков близки к 10%: среднее значение накладных расходов по всем запускам составляет 15%, а медиана — 8%. Только для 6 запусков накладные расходы превышают 20%. На трёх из них Racer работает от 14 до 70 секунд, а RInGen-CICI на 40-50% медленнее из-за накопленного количества одновременно запущенных

[10]Эксперименты были поставлены именно на этой версии, поскольку она даёт то же количество решённых проблем на тестовом наборе по сравнению со стабильной версией, но при этом иногда работает почти в десять раз быстрее

взаимодействующих процессов. Остальные запуски с накладными расходами более 20% — это те, где Racer работает не более 2 секунд, а RInGen-CICI — от 2 до 4 секунд. Отсюда получается высокий процент, которым, по этой причине, можно пренебречь.

Итак, завершая ответ на исследовательский вопрос 2 отметим, что, как было представлено на рисунке 6.3, медиана накладных расходов RInGen-CICI составляет около 8%. Высокие ($> 50\%$) накладные расходы наблюдаются только на шести запусках.

6.4.3 Значимость класса индуктивных инвариантов

Трудно *точно* подсчитать, какие из проблем, решаемых только RInGen-CICI, не входят в класс инвариантов Elem, поскольку задача формального доказательства невыразимости в классе Elem достаточно трудоёмка даже для человека. Однако число таких проблем можно оценить как количество тех проблем, где вызываемый решатель возвращает либо древовидный автомат с циклами, либо насыщение; все уникальные проблемы с результатом «SAT», полученные инструментом RInGen-CICI, подходят под этот критерий. Из этого следует, что все инварианты уникально решённых инструментом RInGen-CICI проблем не принадлежат к классу инвариантов Elem. Таким образом, основная причина успеха инструмента RInGen-CICI по сравнению с другими инструментами заключается в выразительности используемого им класса индуктивных инвариантов.

Заключение

Основные результаты работы заключаются в следующем.

1. Предложен эффективный метод автоматического вывода индуктивных инвариантов, основанных на автоматах над деревьями, при этом данные инварианты позволяют выражать рекурсивные отношения для большого количества реальных программ; метод базируется на поиске конечных моделей.
2. Предложен метод автоматического вывода индуктивных инвариантов, основанный на трансформации программы и поиске конечных моделей, в классе инвариантов, основанном на синхронных автоматах над деревьями; этот класс позволяет выражать рекурсивные отношения и обобщает классические символьные инварианты.
3. Предложен класс индуктивных инвариантов, основанный на булевой комбинации классических инвариантов и автоматов над деревьями, который, с одной стороны, позволяет выражать рекурсивные отношения в реальных программах, а, с другой стороны, позволяет эффективно выводить индуктивные инварианты; также предложен эффективный метод совместного вывода индуктивных инвариантов в этом классе посредством вывода инвариантов в комбинируемых подклассах.
4. Проведено теоретическое сравнение существующих и предложенных классов индуктивных инвариантов; в том числе сформулированы и доказаны леммы о «накачке» для языка ограничений и для языка ограничений расширенного функцией размера терма, которые позволяют доказывать невыразимость инварианта в языке ограничений.
5. Выполнена пилотная программная реализация предложенных методов на языке F# в рамках инструмента RInGen; инструмент сопоставлен с существующими методами на общепринятом тестовом наборе задач верификации функциональных программ «Tons of Inductive Problems»: реализация наилучшего из предложенных методов смогла за отведённое время решить в 3.74 раза больше задач, чем существующие инструменты.

В качестве **рекомендации по применению результатов работы** в индустрии и научных исследованиях следует указать, что разработанные методы

применимы для автоматизации рассуждений о системах дизъюнктов Хорна над теорией алгебраических типов данных, а также что их реализация выполнена в публично доступном инструменте RInGen. Созданный инструмент может быть использован в качестве основной компоненты для верификации в статических анализаторах кода и верификаторах для языков с алгебраическими типами данных, таких как Rust, Scala, Solidity, Haskell и OCaml. Инструмент может быть использован для доказательства недостижимости ошибок или заданных фрагментов кода, что является важным для задач компьютерной безопасности и обеспечения качества.

В качестве **перспективы дальнейшей разработки тематики** можно предложить расширение предложенных классов индуктивных инвариантов и методов их вывода на комбинации алгебраических типов данных с другими типами данных, распространённых в языках программирования, таких как целые числа, массивы, строковые типы данных. Это позволит выводить инварианты программ со сложными функциональными взаимосвязями между структурами и лежащими в них данными, что существенно расширит практическую применимость предложенных методов.

Список литературы

1. Symbolic model checking [Текст] / 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. [Текст] / P. Godefroid, M. Y. Levin, D. Molnar // Queue. — New York, NY, USA, 2012. — Vol. 10, no. 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 [Текст] / M. Wohrer, U. Zdun // 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE). — 2018. — P. 2—8.
4. *Eriksen, M.* Scaling Scala at Twitter [Текст] / M. Eriksen // ACM SIGPLAN Commercial Users of Functional Programming. — Baltimore, Maryland: Association for Computing Machinery, 2010. — (CUFP '10). — URL: https://doi.org/10.1145/1900160.1900170.
5. *Metz, C.* The Epic Story of Dropbox's Exodus From the Amazon Cloud Empire [Электронный ресурс] / C. Metz. — URL: https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire/ (visited on 11/26/2022).
6. *Floyd, R. W.* Assigning meanings to programms [Текст] / R. W. Floyd // Proccedings of the AMS Symposium on Appllied Mathematics. Vol. 19. — American Mathematical Society, 1967. — P. 19—31.
7. *Hoare, C. A. R.* An Axiomatic Basis for Computer Programming [Текст] / C. A. R. Hoare // Commun. ACM. — New York, NY, USA, 1969. — Vol. 12, no. 10. — P. 576—580. — URL: https://doi.org/10.1145/363235.363259.
8. *Rushby, J.* Subtypes for specifications: predicate subtyping in PVS [Текст] / J. Rushby, S. Owre, N. Shankar // IEEE Transactions on Software Engineering. — 1998. — Vol. 24, no. 9. — P. 709—720.
9. Flux: Liquid Types for Rust [Текст] / N. Lehmann [et al.]. — 2022. — URL: https://arxiv.org/abs/2207.04034.

10. *Suter, P.* Satisfiability Modulo Recursive Programs [Текст] / P. Suter, A. S. Köksal, V. Kuncak // Static Analysis / Ed. by E. Yahav. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. — P. 298—315.

11. *Leino, K. R. M.* Dafny: An Automatic Program Verifier for Functional Correctness [Текст] / 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.

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

13. *Müller, P.* Viper: A Verification Infrastructure for Permission-Based Reasoning [Текст] / 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.

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

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

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

17. *Vezzosi, A.* Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types [Текст] / 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.

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

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

20. Synthesizing Software Verifiers from Proof Rules [Текст] / 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.

21. Horn Clause Solvers for Program Verification [Текст] / 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.

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

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

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

25. *K, H. G. V.* Solving Constrained Horn Clauses modulo Algebraic Data Types and Recursive Functions [Текст] / 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.

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

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

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

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

30. *Angelis, E. D.* CHC-COMP 2022: Competition Report [Текст] / 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.

31. Satisfiability of constrained Horn clauses on algebraic data types: A transformation-based approach [Текст] / E. De Angelis [et al.] // Journal of Logic and Computation. — 2022. — Vol. 32, no. 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.

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

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

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

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

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

37. *Kostyukov, Y.* Beyond the Elementary Representations of Program Invariants over Algebraic Data Types [Текст] / 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.

38. *Kostyukov, Y.* Collaborative Inference of Combined Invariants [Текст] / 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.

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

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

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

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

43. *Clarke, E. M.* The Birth of Model Checking [Текст] / 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.

44. *Kautz, H.* Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search [Текст] / 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).

45. Chaff: Engineering an Efficient SAT Solver [Текст] / 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.

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

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

48. *Stump, A.* CVC: A Cooperating Validity Checker [Текст] / 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.

49. Symbolic Model Checking without BDDs [Текст] / 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.

50. *Kurshan, R. P.* The Automata-Theoretic Approach [Текст] / R. P. Kurshan. — Princeton: Princeton University Press, 1995.

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

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

53. *McMillan, K. L.* Applications of Craig Interpolants in Model Checking [Текст] / 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.

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

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

56. IC3 Modulo Theories via Implicit Predicate Abstraction [Текст] / 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.

57. *Hoder, K.* Generalized Property Directed Reachability [Текст] / 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.

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

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

60. *Blass, A.* Existential fixed-point logic [Текст] / 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.

61. *Blass, A.* The Underlying Logic of Hoare Logic [Текст] / 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/.

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

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

64. *Mordvinov, D.* Synchronizing Constrained Horn Clauses [Текст] / 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.

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

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

67. *Cousot, P.* Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints [Текст] / 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.

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

69. *Giacobazzi, R.* Analyzing program analyses [Текст] / R. Giacobazzi, F. Logozzo, F. Ranzato // ACM SIGPLAN Notices. — 2015. — Vol. 50, no. 1. — P. 261—273.

70. *Cousot, P.* Abstract Interpretation Frameworks [Текст] / P. Cousot, R. Cousot // Journal of Logic and Computation. — 1992. — Vol. 2, no. 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.

71. *Campion, M.* Partial (In)Completeness in Abstract Interpretation: Limiting the Imprecision in Program Analysis [Текст] / 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.

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

73. *Moura, L. de.* Z3: An Efficient SMT Solver [Текст] / 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.

74. cvc5: A Versatile and Industrial-Strength SMT Solver [Текст] / 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.

75. *Rümmer, P.* A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic [Текст] / 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.

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

77. *Xi, H.* Dependent Types in Practical Programming [Текст] / 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.

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

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

80. *Hamza, J.* System FR: Formalized Foundations for the Stainless Verifier [Текст] / 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.

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

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

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

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

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

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

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

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

89. *Torlak, E.* Kodkod: A Relational Model Finder [Текст] / 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.

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

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

92. *Reger, G.* Finding Finite Models in Multi-sorted First-Order Logic [Текст] / 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.

93. *Lisitsa, A.* Finite Models vs Tree Automata in Safety Verification [Текст] / 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.

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

95. *Oppen, D. C.* Reasoning about Recursively Defined Data Structures [Текст] / 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.

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

97. *Schulz, S.* E - a Brainiac Theorem Prover [Текст] / S. Schulz // AI Commun. — NLD, 2002. — Vol. 15, no. 2, 3. — P. 111—126.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

115. *Veanes, M.* Symbolic tree automata [Текст] / M. Veanes, N. Bjørner // Information Processing Letters. — 2015. — Vol. 115, no. 3. — P. 418—424.

116. *D'Antoni, L.* Minimization of Symbolic Tree Automata [Текст] / 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.

117. *Faella, M.* Reasoning About Data Trees Using CHCs [Текст] / M. Faella, G. Parlato // Computer Aided Verification / Ed. by S. Shoham, Y. Vizel. — Cham: Springer International Publishing, 2022. — P. 249—271.

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

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

120. *Reynolds, A.* Induction for SMT Solvers [Текст] / 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.

121. *Yang, W.* Lemma Synthesis for Automating Induction over Algebraic Data Types [Текст] / 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.

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

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

124. Decidability of Inferring Inductive Invariants [Текст] / 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.

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

126. *Losekoot, T.* Automata-Based Verification of Relational Properties of Functions over Algebraic Data Structures [Текст] / 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.

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

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

129. Transition Power Abstractions for Deep Counterexample Detection [Текст] / 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.

130. Predicting Rankings of Software Verification Tools [Текст] / 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.

Список листингов кода

Список рисунков

Список таблиц

Saint Petersburg State University

On the rights of the manuscript

Kostyukov Yurii

Automatic Inference of Inductive Invariants of Programs With Algebraic Data Types

Scientific specialty
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, Professor
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]. For example, Twitter uses Scala for most of its server applications [4], while Dropbox uses Rust for managing data flows [5], and in both cases algebraic data types are actively employed.

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

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 [6; 7] or refinement types [8], as, for example, in FLUX system [9] for RUST and LEON system [10] for 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 [11], WHY3 [12], VIPER [13], F* [14], face the same problem. It should also be noted that algebraic data types underlie numerous interactive theorem provers (ITPs), such as COQ [15], IDRIS [16], AGDA [17], LEAN [18]. 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 [19].

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 [20; 21]. For instance, RUSTHORN [22] utilizes Horn solvers for verifying RUST programs, while SOLCMC [23] is employed for verifying SOLIDITY smart contracts.

There are efficient Horn solvers with ADT support, such as SPACER [24] and its descendant RACER [25], as well as ELDARICA [26], HOICE [27], RCHC [28], VERICAT [29]. Annual international competitions CHC-COMP [30] 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* [21], 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 [26]. 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) [25]. 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 [31–34]. 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 [29]. 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) [28] to express inductive invariants of programs over ADTs using *tree automata* [35]. However, due to the problems with representing tuples of

terms using tree automata, the proposed class of invariants does not include classical symbolic invariants. As a result, this approach is often inapplicable for 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 relations and include classical symbolic invariants.
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 relations and generalizes classical symbolic invariants.
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 [36], 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 [37], 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 [38], 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 [39], 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 130 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) [40] and Rice's theorem (1953) [41]. 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) [6] and C. A. R. Hoare (1969) [7]. 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 [42]. Its essential limitation was the so-called state explosion problem [43]: 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 [44]. 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 [45], based on the Conflict Driven Clause Learning (CDCL) algorithm for satisfiability checking [46]. 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 [47];

it was designed specifically for formal methods problems. In 2002, the first SMT solver CVC [48] 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 [49]. 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 [50; 51]. 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 [52; 53]. 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 [20]. 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 [54]. ICE is implemented in the Horn solvers HOICE and RCHC.

In 2011, A. R. Bradley proposed an approach called IC3/PDR (property-directed reachability) [55] for SAT-based hardware verification. By 2014, the approach was generalized for SMT-based software verification [56; 57]. 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 [24] and RACER [25].

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 [6; 7], 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 [58] 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 [59]. Therefore, in 1987 A. Blass and Yu. Gurevich proposed to abandon first-order logic in favor of *existential fixed-point logic* [60; 61]. 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 [21]. 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 [31–34; 62; 63]), syntactic synchronizations of clauses [63; 64], to the inference of relational invariants (invariants for several predicates) [65; 66].

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 [67] 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 [68]. However, as shown by R. Giacobazzi et al. [69] 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 [70]. Recent works in the field [71; 72] 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 [73], CVC5 [74] and PRINCESS [75], and by automated theorem provers (ATPs) such as VAMPIRE [76]. 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 [21]. 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 [26] 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 [20; 21]. Among approaches providing such a reduction, the most significant are the Floyd-Hoare logic for imperative programs [6; 7], as well as dependent types [77] and refinement types [8] for functional programs. There are many tools within which this reduction can be implemented, for example, LiquidHaskell [78] for the Haskell language, RCaml [79] for OCaml, Flux [9] for Rust, Leon [10] and Stainless [80] for the Scala language. For example, tools like RustHorn [22], a verifier for the Rust language, and SolCMC [23], 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 [81–86]. 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 [35].

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 [87]. 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 [88], KODKOD [89], PARADOX [90], as well as CVC5 [91] and VAMPIRE [92] 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 [93], as well as in first-order infinite model building [94].

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 [37].

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 [95], 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 [21]. 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 [96], E [97] and ZIPPERPOSITION [98], 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 [99]. 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.

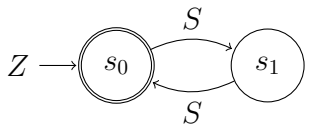
2.3 Regular Invariants

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.

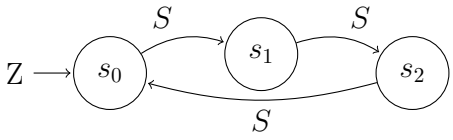
$$
\begin{aligned}
x = Z \wedge y = S(Z) &\rightarrow inc(x, y) \\
x = S(x') \wedge y = S(y') \wedge inc(x', y') &\rightarrow inc(x, y) \\
x = S(Z) \wedge y = Z &\rightarrow dec(x, y) \\
x = S(x') \wedge y = S(y') \wedge dec(x', y') &\rightarrow dec(x, y) \\
inc(x, y) \wedge dec(x, y) &\rightarrow \bot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

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 [35].

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

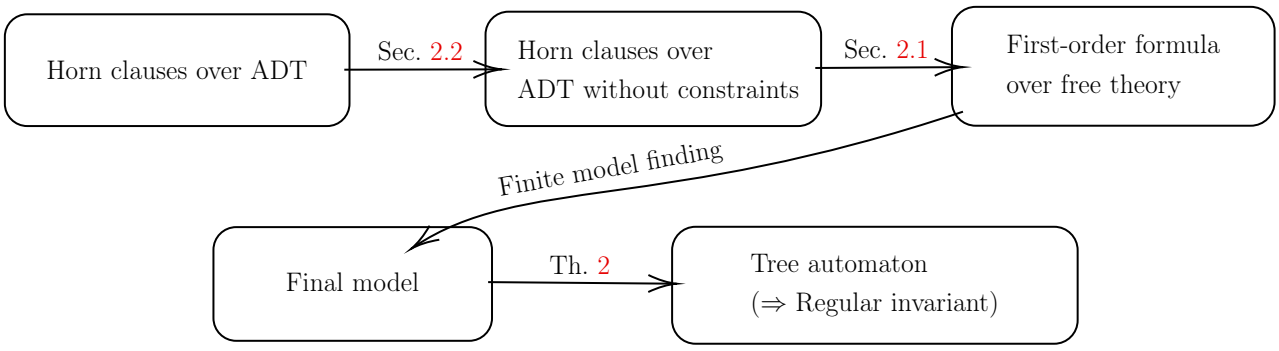
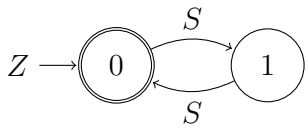


Figure 2.1 – Regular invariant inference method for a Horn clause system over ADT

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 [88], Kodkod [89], Paradox [90], and general theorem provers, such as cvc5 [91] and Vampire [92], 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 [96], E [97] and Zipperposition [98] 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 [88], Kodkod [89], Paradox [90], and even cvc5 [91] and Vampire [92] 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. 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 [35] and full [28] 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. [28; 35]). 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. [28]). 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 [35, 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 [35, 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) [51] 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 [38].

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. [6]). 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* [100] 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 [67] 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.

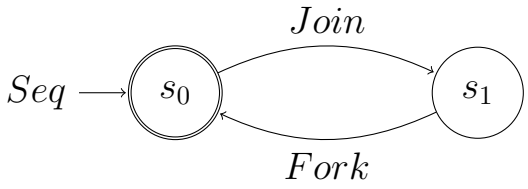
This program is safe with respect to the property but it has no inductive invariants expressible in the ELEM or REG classes. However, it has combined invariants in ELEM $\uplus$ REG. That is, for any programs $p, t : Prog$ function $\texttt{ok}(p)$ returns $\texttt{true}$ iff the following holds:

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

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 [101]: 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 [51], 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 [96] and finite-model finders [90; 91].

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 $\langle \mathcal{S}, Init, T \rangle$:

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

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 $\langle \mathcal{S}, Init, T \rangle$ 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 [24] and HoIce [27]), elementary invariants with term size constraints (SizeElem, inferred by Eldarica [26]), regular invariants (Reg, inferred by RCHC [28], as well as the method from Chapter 2), synchronous regular invariants (Reg_+, $\text{Reg}_\times$, inferred by RCHC [28], 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 [37].

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 [35, property 3.2.9]
[2] see Section 3.1.2
[3] see [95]
[4] see [102]
[5] see [35, Section 3.2.1 and Th. 1.7.2]
[6] see [35, Section 3.2.1 and Th. 1.7.4]
[7] see [35, Def. 3.2.1 and Th. 1.7.2]
[8] see [35, Def. 3.2.1 and Th. 1.7.4]
[9] see Section 3.1.3
[10] see [103, 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] $\text{REG} \subseteq \text{REG}_+$ [35, Prop. 3.2.6]
[5] $\text{REG}_+ \subseteq \text{REG}_\times$ [28, Th. 11]

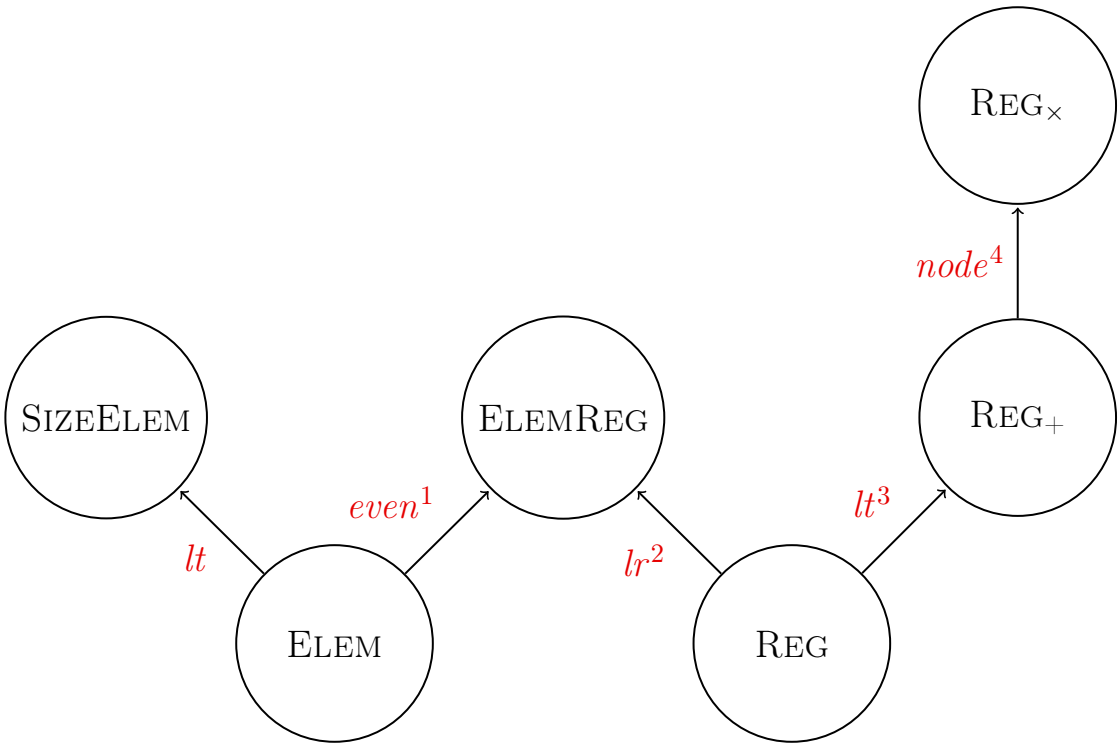


Figure 5.1 – Inclusion relations between classes of inductive invariants over ADTs.

Reg+ and Reg× 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 [35, 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 [35, 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 [104, 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 [105] 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 [95]). 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 $\mathit{leaves}_\sigma(g) \triangleq \{p \mid p\,(g)$ 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 $\mathcal{H}\mathit{eight}(g_i) > K$,
- 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 $\mathcal{H}\mathit{eight}(t) > N$ 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 [37]. □

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.

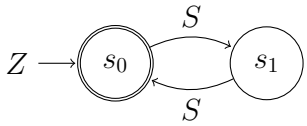
Theorem 19. There are regular but non-elementary invariants, i.e., $\textsc{Reg} \setminus \textsc{Elem} \neq \varnothing$.

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 [106]). Any $\textsc{SizeElem}$-formula is equivalent to some quantifier-free $\textsc{SizeElem}*$-formula.

Definition 30. Borrowing notation from [102], we denote $\mathbb{T}_{\sigma}^{k} = \{t \text{ has sort } \sigma \mid size(t) = k\}$. 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 $\mathbb{T}_{\sigma}^{b'} \neq \varnothing$, then $\left|\mathbb{T}_{\sigma}^{b'}\right| \geqslant n$ 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 [37]. □

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 $Tree ::= left \mid node(Tree, Tree)$, which checks whether the number of

nodes in the leftmost branch of the tree is even.

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

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* $\in$ Reg $\setminus$ 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. $\square$

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 [107]. 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 [108–111]. 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 [112; 113]. 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 [114], and, in particular, there are numerous works proposing the integration of various types of synchronization into tree automata [81–86]. 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 [115; 116]. 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 [117]. 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 have been implemented in a Horn solver RInGen (*R*egular *In*variant *Gen*erator)[1], which was developed as part of this thesis. The implementation comprises 5200 lines of F# code. The decision was made to create a Horn solver from scratch, rather than integrating into the codebases of existing solvers, as the 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 Horn clause system with constraints in the SMTLIB2 format [118]. After the clauses are parsed, they are simplified, in particular, equalities, selectors, and testers are eliminated. Then, depending on the options submitted to the Horn solver, one of the algorithms proposed in this work is initiated. The output of each of these algorithms is a formula over uninterpreted functions, which is passed to an external logic solver — either Vampire or cvc5. As a result, the tool returns a safe inductive invariant if the system is safe, otherwise, it returns a resolution refutation.

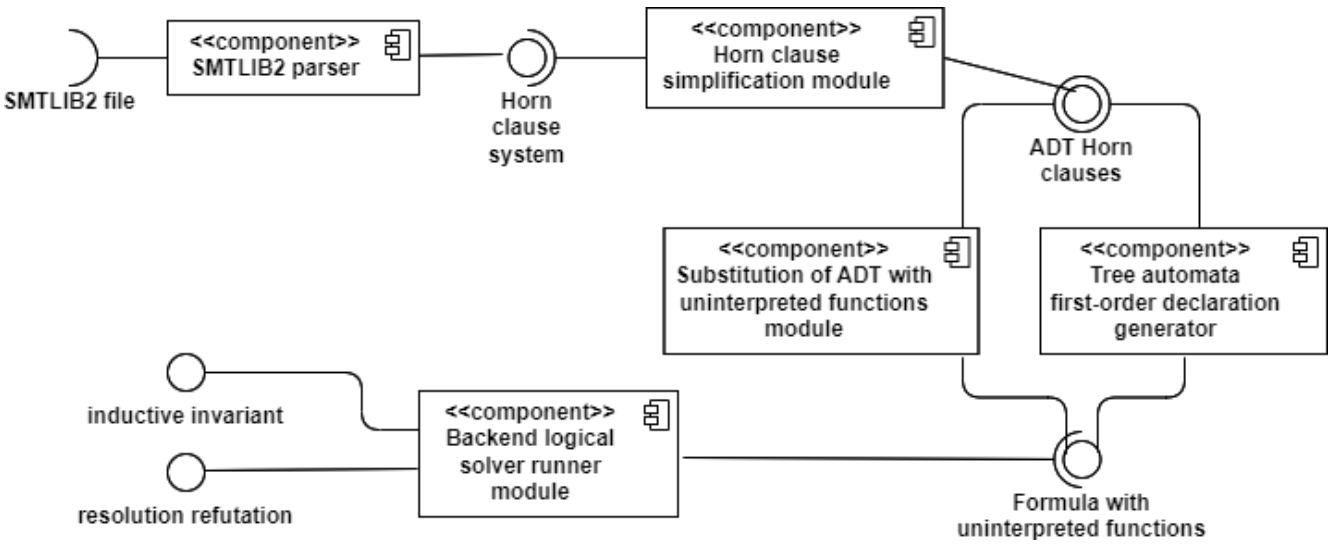


Figure 6.1 – Architecture of RInGen

RInGen. Hence, we implemented the approach presented in Chapter 2 within the Horn solver RInGen. Both the approach itself and its implementation involve the use of an external SMT solver $\mathcal{V}$ for the theory of uninterpreted

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

functions with quantifiers, therefore the proposed implementation will be further denoted as RInGen($\mathcal{V}$). Specifically, Vampire [76] and the SMT solver cvc5 are used as the $\mathcal{V}$ in the experiments. Vampire uses a portfolio-based approach [119], that is, it iterates through various formula satisfiability checking techniques, based on saturation of the system [96] or finite model finding [92]. cvc5 is used in the finite model finding mode[2] [91]. Both tools allow for both proving the safety of the system and finding counterexamples.

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

RInGen-CICI. The approach presented in Chapter 4 has been implemented within the codebase of the Horn solvers Racer[25] (a development of the Horn solver Spacer[24], implemented in the logical solver Z3[5]) and the Horn solver RInGen($\mathcal{V}$)[6], which was described earlier. This implementation will be referred to as RInGen-CICI($\mathcal{V}$) in the future. The following describes both parts of this implementation in the Horn solvers Racer and RInGen($\mathcal{V}$), respectively, which are subsequently referred to as *basic* relative to the tool RInGen-CICI($\mathcal{V}$).

Z3/Racer. The Racer Horn solver was 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) [24], which can be considered a complex instance of CEGAR. PDR builds abstract states in the form of conjunctions of formulas (referred to as *lemmas*) at various *levels* by iteratively increasing the level in a loop. The following capabilities are supported: if a set of lemmas $\{\varphi_i\}$ was constructed at level n, then $\bigwedge_i \varphi_i$ overapproximates all states reachable in less than n transition steps, and under-approximates the safety property. Thus, in PDR, lemmas fulfill the requirement of abstraction in the Collaborate procedure (Listing 4.4). As part of this thesis, Racer was modified to asynchronously

[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

pass the set of lemmas from the last level to a new process of RInGen($\mathcal{V}$) at the end of each iteration.

RInGen($\mathcal{V}$). The Collaborate procedure (Listing 4.4) is implemented based on the RInGen($\mathcal{V}$). The following generalization was carried out in the ResidualCHCs($\mathcal{P}, a$) procedure (see 4.2.3). The conjunctive form of lemmas from Racer is used to infer more general invariants: $\bigwedge_i(\varphi_i(\overline{x}) \vee \overline{x} \in L_i)$. 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 modified RInGen($\mathcal{V}$) calls the external solver $\mathcal{V}$ with a time limit of 30 seconds. Then its results are passed back to Racer, where they are processed asynchronously. Moreover, the implementation does not perform the costly transformation into CNF from Listing 4.5, as the RInGen($\mathcal{V}$) implementation allows inputting 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 of this chapter 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 [120], AdtInd [121] and others were not considered, as they do not accept Horn clause systems as input. Also, logic programming tools (such as Prolog [122]) were not considered because they only check the unsatisfiability of Horn clause systems and say nothing about 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 Chapter 6, and the methods they implement are described in Chapters 2, 3, and 4 of this work. In the table, for brevity, the name RInGen is

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
R(cvc5)	Reg	Transf. + FMF	Yes	Yes
R(Vampire)	–	Transf. + Saturation	No	Yes
R-Sync	$\text{Reg}_{\times}$	Transf. + FMF	Yes	Yes
R-CICI(cvc5)	ElemReg	CEGAR($\mathcal{O}$)	Yes	Yes
R-CICI(Vampire)	–	CEGAR($\mathcal{O}$)	No	Yes

abbreviated to R, so for example, the Horn solver RInGen-Sync in the table is represented as R-Sync. 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 [91; 92]); a dash in the "Class of Invariants" column means the following: although when the system is satisfiable, the output of the tool implicitly encodes its inductive invariant, there is no everywhere-halting procedure that allows this output to be checked. The other designations are explained in the subsections dedicated to the corresponding tools.

Most of the tools examined differ in the classes in which they seek inductive invariants. A comparison of these classes of invariants was given in Chapter 5. This comparison is important for comparing tools for the following reason: if a tool outputs invariants in a certain class, then the problem of expressiveness 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 check

whether there is an invariant *at all* for a given Horn clause system in its class[7], then in the absence of such, the tool will not terminate.

Further on, we provide a short comparartive descrition of the existing tools.

The SPACER tool [24] 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 [125]. At its core, the tool employs the SPACER approach, which is based on a technique called *property-directed reachability* (IC3/PDR), evenly distributing analysis time between counterexample search and safe inductive invariant construction, propagating information about reachability of unsafe properties and partial safety lemmas. The tool allows for the output of 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 [25] is an evolution of the SPACER tool, allowing for the output of invariants in the constraint language expanded with catamorphisms. This constraint language is denoted in table 6.1 as CATELEM. RACER also inherits all the advantages of the SPACER approach. A drawback of the approach is that it's not fully automatic, as it requires manually describing catamorphisms, which can be challenging in practice because it can be difficult to understand which catamorphisms will be required for the invariant based on the given problem. A drawback of the tool itself is that it does not return any verifiable certificates with catamorphisms.

The ELDARICA tool [26] constructs models with term size constraints, which calculate the total number of constructor occurrences (the SIZEELEM class). This extension only marginally enhances the expressiveness of the constraint language, as the introduced function counts all constructors at once, thus it cannot express many properties, such as a restriction on tree height. The ELDARICA tool employs the CEGAR approach with predicate abstraction and an embedded SMT solver PRINCESS[75], which provides a resolution procedure, as well as an interpolation procedure for algebraic data types with term size constraints. These procedures are

[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, [123; 124])

built on the reduction of this theory to a combination of theories of uninterpreted functions and linear arithmetic[102].

The VeriCaT tool [31–34] processes verification conditions 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 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 also 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 [27] constructs elementary invariants using an approach based on machine learning, ICE [54]. 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 expressive constraint language, and thus often does not terminate.

The RCHC tool [28; 126] also uses the ICE approach; it is based on machine learning, but expresses inductive invariants of programs over ADTs using *tree automata* [35]. However, due to the complexities of expressing tuples of terms with automata, described in Section 5.2.1, the approach often proves inapplicable for even the simplest examples where classical symbolic invariants exist.

Conclusions. Summarizing the comparison, the following can be said. Compared with 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 to the inductive invariant of the system, if it exists. Compared with the methods of the remaining existing tools, the proposed approaches allow inferring invariants in independent classes of regular invariants. Therefore, the application of the proposed methods in conjunction with existing ones will solve a wider variety of problems.

6.3 Evaluation

6.3.1 Tool Selection

We have selected the RACER [25] and ELDARICA [26] tools for compassion, which are Horn solvers with support for algebraic data types leading in the CHC-COMP competition [30]. We also selected CVC5-IND (CVC5 in induction mode) [120] and VERICAT [31]. Although these tools do not build inductive invariants explicitly, making it impossible to check their correctness, their launch on an equivalent benchmark is added to the experimental comparison as they solve a related task.

6.3.2 Benchmark Suite

The experiments were conducted on the TIP (Tons of Inductive Problems) benchmark suite [127], the test set from the 2022 CHC-COMP competition track[8], dedicated to algebraic data types. The TIP set consists of 454 Horn clause systems derived from Haskell programs with ADTs and recursion. The test set includes the following algebraic data types: lists, queues, regular expressions, and Peano integers.

6.3.3 Setup

The experiments were conducted on the StarExec platform [128], which has a cluster of machines with Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz and Red Hat Enterprise Linux 7[9], with a limit on the CPU runtime for each tool on each test of 600 seconds and a memory limit of 16 GB.

6.3.4 Research Questions

We have formulated the following research questions to guide the experimental setup.

Research question 1 (Number of solutions).

- Do the proposed methods allow for the verification of more systems than approaches that build classical symbolic invariants?
- Do the proposed methods allow for the verification of systems that have classical invariants?

[8] https://github.com/chc-comp/ringen-adt-benchmarks

[9] https://www.starexec.org/starexec/public/machine-specs.txt

Table 6.2 – Results. SAT indicates that the system is safe (there is an inductive invariant), UNSAT indicates that the system is unsafe.

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

Research question 2 (Performance).

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

Research question 3 (Significance of the inductive invariant class). Collaborative inference, implemented in RInGen-CICI, theoretically allows not only to infer invariants in a larger class but also to accelerate the convergence of the search for classical symbolic invariants. What is the share of classical symbolic invariants in all uniquely solved problems 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 instruments is presented in Table 6.2.

RInGen. On all 12 problems where Eldarica returned UNSAT, RInGen terminated with the same result, and it found more counterexamples. The RInGen(cvc5), RInGen(Vampire), and Racer found counterexamples for 21, 46, and 22 clause systems, respectively. Most of these systems are the same, although some are unique to each tool. RInGen(Vampire) constructed significantly more

UNSAT results than other tools, as Vampire implements an efficient refutation inference procedure. Therefore, even though the proposed algorithms are designed to search for more inductive invariants, they also allow for finding 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 solved 25 unique (not solved by RInGen(cvc5)) tasks, each of which is a formulation of some property of ordinal predicates ($<, \leqslant, >, \geqslant$) on Peano numbers. These problems are easily solved by Eldarica, as ordinal predicates are themselves included in the SizeElem verification domain at the primitive level. However, RInGen(cvc5) solved 13 unique (not solved by Eldarica) problems, which invariants are not expressible in the verification domain of the Eldarica. The effectiveness of the approach implemented in the RInGen heavily depends on the external solver used, as evidenced by the fact that RInGen(Vampire) inferred more than 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 were 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 fully-convolutional synchronous tree automata used in RInGen-Sync. Racer concluded 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 fully-convolutional synchronous tree automata used in RInGen-Sync, which should yield a larger number of solutions, the finite model finding tools that RInGen-Sync uses as a backend do not terminate on problems with a large number of quantifiers, and therefore RInGen-Sync often does not terminate. The results do not change when changing the backend from cvc5 to other finite model finding tools and increasing the time limit to 1200 seconds. The small overlap with Racer suggests that although in theory the class of elementary invariants is 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 solved fewer *unsafe problems* than the best of the basic solvers: RInGen-CICI obtained 19 (with cvc5) and 28 (with Vam-

PIRE) 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 [129]. Thus, all counterexamples obtained by RINGEN-CICI are directly obtained from one of the basic solvers. Some counterexamples found by RINGEN were 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, were also obtained by RINGEN-CICI, with the exception of one UNSAT answer.

On safe problems, RINGEN-CICI outperformed competitive 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 allows to establish the feasibility of significantly more systems than the parallel launch of basic tools.

However, there are problems that were solved by the basic solvers but not by the proposed tool. RINGEN-CICI(CVC5) did not solve 7 problems that were 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 was increased. Existing methods of prediction of verification time, such as [130], can be applied to avoid hard-coding a time limit at all. 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 an increase in overhead in the

tool. A similar situation is observed 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.

Therefore, RInGen(cvc5) did not outperform the existing solutions, but it did provide many unique solutions compared to them, as it inferred invariants in a new class. RInGen(Vampire), built on the same approach, solved over 2.5 times more problems than the best of the existing tools, for the same reason. Despite the fact that the class of invariants of RInGen-Sync is significantly broader, inferring invariants in it is much more labor-intensive, so although it was able to infer almost twice as many invariants as RInGen(cvc5), it did not outperform the best of the existing tools. The best results were shown by RInGen-CICI(cvc5), which solved 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 size of the class of invariants and the efficiency of the invariant inference procedure. The best of the proposed tools, RInGen-CICI(Vampire), solved a total of 189 + 28 problems, which is about 3.74 times more than the best of the existing tools, Eldarica, which solved 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, on average, by one order of magnitude. In the figure, some unsafe systems were checked faster by cvc5-Ind, VeriCaT, and Racer. This may be associated with 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). On problems that were solved by several tools, the RInGen(cvc5) operates on average two orders of magnitude faster. The RInGen(Vampire) runs took even less time than RInGen(cvc5).

RInGen-Sync and RInGen-CICI. Figure 6.4 presents the *overall performance plots* for RInGen-CICIcompared 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 errors of the tools, which occurred for both Racer and RInGen-CICI, due to the instability of the used version of

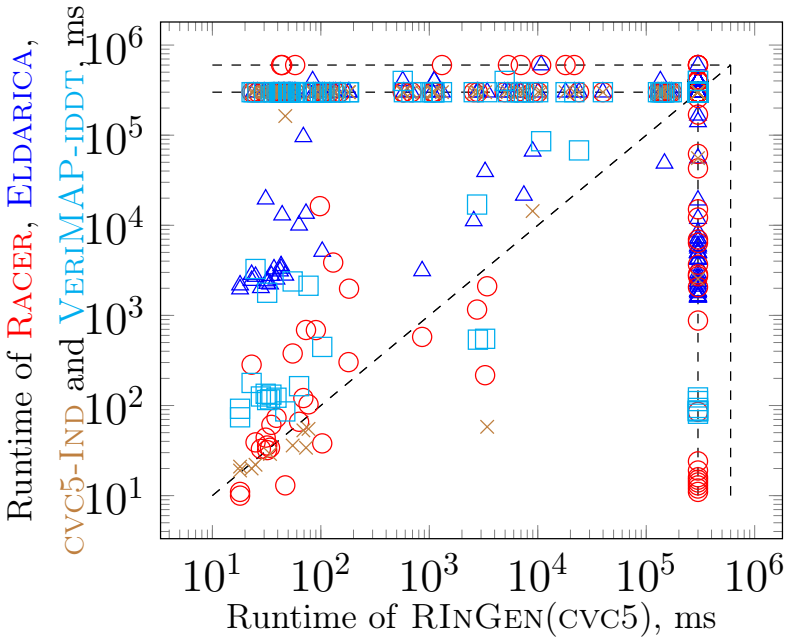


Figure 6.2 – Performance comparison. Each point on the plot represents a pair of runtimes.

Racer[10]. The inner dashed lines denote cases when the solver reached the time limit. Since RInGen-CICI solved 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 joint operation finished after the first joint 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 did not hit the dashed lines are near the diagonal on both plots, meaning that RInGen-CICI worked comparably with Racer on problems that were solved by both tools.

The graph 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 was faster than Racer, but in the remaining 34, no backend call was successful, so RInGen-CICI behaved just like Racer, but with the overhead of process creation. The overhead for these 34 runs is shown

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

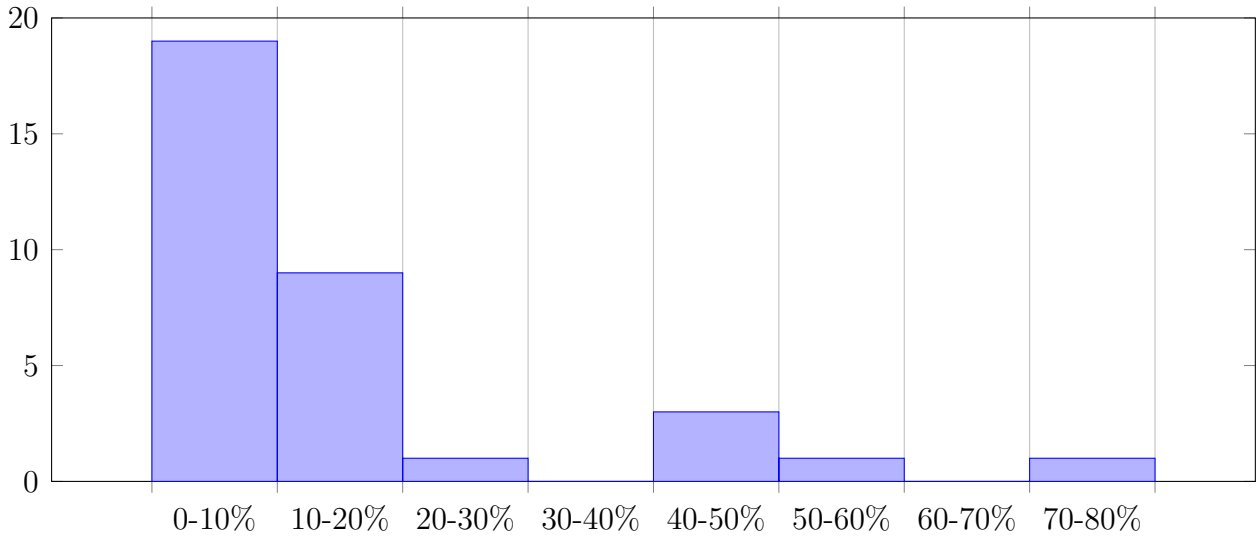


Figure 6.3 – Number of benchmark instances (y-axis) solved by both RInGen-CICI and Racer, and the CPU time overhead (x-axis) of running RInGen-CICI compared to Racer. Racer outperforms RInGen-CICI on 34 instances. There are no instances with an overhead larger than 80%, so the x-axis is not shown further.

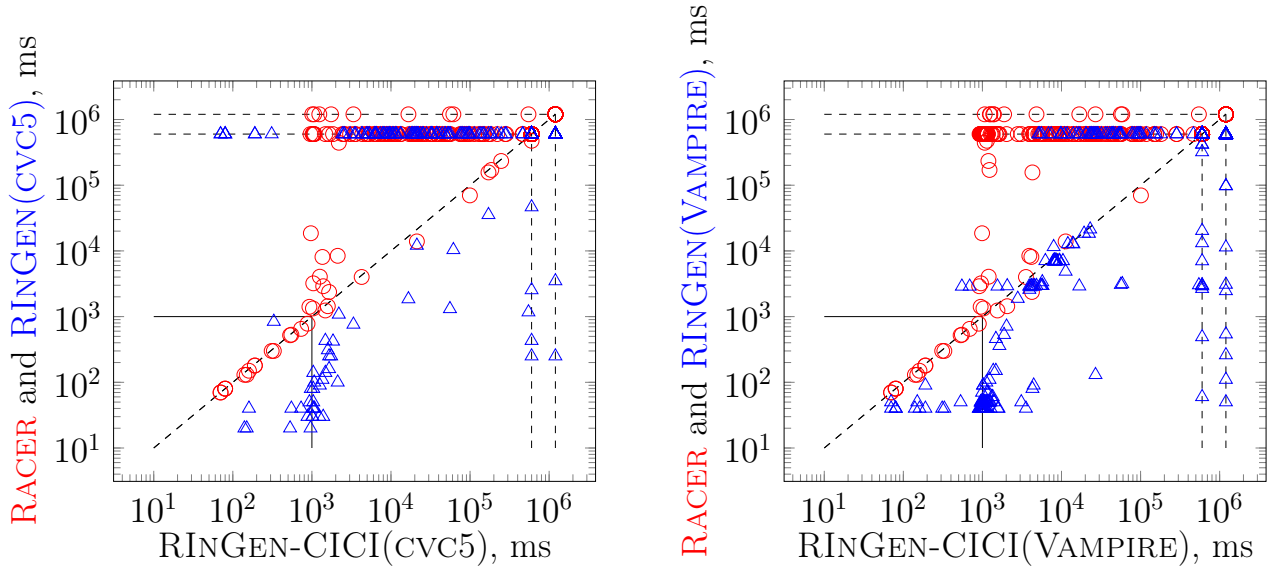


Figure 6.4 – Runtime comparison

in Figure 6.3. The chart shows how many times slower RInGen-CICI operates 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, for this reason, can be disregarded.

Concluding the answer to research question 2, we note that, as presented in Figure 6.3, 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 proving inexpressibility in Elem is fairly labor-intensive even for a human. However, the number of such problems can be estimated as the count 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 relations and generalizes classical symbolic invariants.
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. *Eriksen, M.* Scaling Scala at Twitter [Text] / M. Eriksen // ACM SIGPLAN Commercial Users of Functional Programming. — Baltimore, Maryland: Association for Computing Machinery, 2010. — (CUFP '10). — URL: https://doi.org/10.1145/1900160.1900170.
5. *Metz, C.* The Epic Story of Dropbox's Exodus From the Amazon Cloud Empire [Electronic Resource] / C. Metz. — URL: https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire/ (visited on 11/26/2022).
6. *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.
7. *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.
8. *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.
9. Flux: Liquid Types for Rust [Text] / N. Lehmann [et al.]. — 2022. — URL: https://arxiv.org/abs/2207.04034.

10. *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.

11. *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.

12. *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.

13. *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.

14. 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.

15. 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.

16. *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.

17. *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.

18. *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.

19. *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.

20. 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.

21. 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.

22. *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.

23. 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.

24. *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.

25. *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.

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

27. *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.

28. *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.

29. 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.

30. *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.

31. 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.

32. 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.

33. 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.

34. 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.

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

36. 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.

37. *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.

38. *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.

39. 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.

40. 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.

41. *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).

42. *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.

43. *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.

44. *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).

45. 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.

46. *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.

47. *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.

48. *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.

49. 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.

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

51. 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.

52. *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.

53. *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.

54. 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.

55. *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.

56. 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.

57. *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.

58. *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.

59. *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.

60. *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.

61. *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/.

62. 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.

63. 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.

64. *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.

65. *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.

66. *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.

67. *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.

68. *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.

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

70. *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.

71. *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.

72. 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.

73. *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.

74. 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.

75. *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.

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

77. *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.

78. 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.

79. *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.

80. *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.

81. *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.

82. *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.

83. *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.

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

85. *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.

86. *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.

87. *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.

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

89. *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.

90. *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.

91. 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.

92. *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.

93. *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.

94. *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.

95. *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.

96. *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.

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

98. *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.

99. *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.

100. 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.

101. 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.

102. *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.

103. *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.

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

105. *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.

106. *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.

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

108. *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.

109. *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.

110. *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.

111. *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.

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

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

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

115. *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.

116. *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.

117. *Faella, M.* Reasoning About Data Trees Using CHCs [Text] / M. Faella, G. Parlato // Computer Aided Verification / Ed. by S. Shoham, Y. Vizel. — Cham: Springer International Publishing, 2022. — P. 249–271.

118. *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/.

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

120. *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.

121. *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.

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

123. 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.

124. 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.

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

126. *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.

127. 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.

128. *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.

129. 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.

130. 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