Синтез арифметико-логических выражений с использованием SMT-решателя

С. В. Козлов, М. Д. Черепанов

Аннотация – В статье исследуется метод автоматического синтеза арифметико-логических программ на основе SMT-решателя Z3. Рассматривается подход Syntax-Guided Synthesis (SyGuS) применительно к безцикловым программам, представляемым в виде ациклических графов вычислений. Описывается формализация задачи синтеза через систему логических ограничений в теории битовых векторов, включающую операционную семантику узлов, структурные ограничения на поток данных и симметрия-устраняющие условия для коммутативных операций. Представлена практическая реализация синтезатора на языке C# с использованием библиотеки Microsoft.Z3, кодирующая структуру и семантику программы как задачу оптимизации с минимизацией функции стоимости. Экспериментальная часть демонстрирует синтез различных целевых функций с двумя-четырьмя входными переменными: время работы составляет от 3 до 227 секунд в зависимости от сложности выражения. Показано, что синтезатор способен автоматически обнаруживать неочевидные алгебраические эквивалентности и генерировать оптимальные по метрике стоимости реализации. Обсуждаются вопросы масштабируемости подхода и экспоненциальный рост сложности при увеличении количества вычислительных узлов. Результаты подтверждают применимость SMT-технологий для задач супероптимизации небольших программных фрагментов с формальными гарантиями корректности.

Ключевые слова – *синтез программ, формальная верификация, Satisfiability Modulo Theories, SMT-решатель, Z3, Syntax-Guided Synthesis, битовые векторы, булево-арифметические выражения, автоматическая оптимизация программ, программирование.*

I. ВВЕДЕНИЕ

Автоматический синтез программ на основе ограничений представляет собой один из перспективных методов разработки надежного кода. Идея заключается в том, чтобы сформулировать требуемую функцию как спецификацию (например, отношения между входными и выходными переменными) и затем с помощью логических решателей SMT (Satisfiability Modulo Theories) найти программу, удовлетворяющую этой спецификации [1, 2, 3].

Такой подход обеспечивает корректность по построению: результат синтеза гарантированно согласован с образцами или формальными спецификациями. Особенно он эффективен при синтезе программ, манипулирующих битовыми векторами и арифметическими операциями, где пространство возможных комбинаций операций огромно, а исчерпывающий ручной подбор выражений практически невыполним [4, 5]. Такие задачи часто возникают в криптографии, оптимизации компиляторов, разработке низкоуровневых алгоритмов обработки данных и реверс-инжиниринге [6].

В настоящей работе мы применяем SMT-подход с целью автоматического синтеза коротких арифметико-логических выражений, соответствующих заданному набору входных и выходных примеров. Примером целевой функции может служить выражение $f(x,y) = (x \oplus y) + (x \wedge y)$, которое SMT-синтезатор должен «восстановить» или «выразить» через набор базовых операций ($\wedge, \oplus, \vee, \ll$ и др.) с минимальными вычислительными затратами и оптимальной длиной результирующего выражения. Задача осложняется тем, что количество возможных комбинаций операций растет экспоненциально с увеличением глубины выражения, что требует эффективных методов поиска и оптимизации.

Основная цель исследования – показать применимость метода синтеза на основе Z3 для поиска эффективных эквивалентных программ. Использование SMT-решателя Z3 позволяет учитывать сложные логические зависимости между операторами в дереве выражения и проводить оптимизацию программы через минимизацию заданной функции стоимости [7, 8]. Данная работа включает теоретический обзор ключевых идей синтеза программ в контексте SMT, описание практической реализации синтезатора, а также экспериментальную оценку на наборе примеров.

Статья получена 10 марта 2026.

Козлов Сергей Валерьевич, Смоленский государственный университет, доцент кафедры прикладной математики и информатики, кандидат педагогических наук, доцент (email: svkozlov1981@yandex.ru)

Черепанов Михаил Денисович, Смоленский государственный университет, студент физико-математического факультета (email: mrflashstudio@gmail.com)

II. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ СИНТЕЗА ПРОГРАММ С ПОМОЩЬЮ SMT

Синтез программ представляет собой задачу автоматического поиска программы, которая соответствует заданным критериям или спецификации. Эти критерии могут быть выражены различными способами: через логические предикаты, описывающие желаемое поведение программы, через набор примеров входных и выходных значений, или через формальные спецификации на специализированных языках. В отличие от традиционной разработки программного обеспечения, где программист вручную конструирует алгоритм, синтез программ предполагает автоматическое построение исполняемого кода на основе декларативного описания требований [9, 10].

Классическим подходом к решению данной задачи является метод Syntax-Guided Synthesis (SyGuS), в котором помимо функциональных требований задаются синтаксические ограничения на форму искомой программы [11, 12]. Эти ограничения определяют грамматику допустимых конструкций, что позволяет существенно снизить пространство поиска и делает задачу вычислимой на практике. Вместо перебора всех возможных программ система ищет решение только среди тех конструкций, которые соответствуют заданной грамматике. Такой подход особенно эффективен, когда известна общая структура целевой программы или набор допустимых операций.

В рамках данной работы мы используем специализированную разновидность этого подхода, ориентированную на синтез безцикловых (прямолинейных) программ. Целевая программа представляется в виде композиции элементарных компонент – базовых операций, таких как арифметические действия, побитовые операции и логические функции. Каждая операция соответствует узлу в вычислительном графе, а связи между узлами определяют поток данных от входов к выходу. Решение задачи синтеза сводится к построению соответствующей логической формулы и её передаче в решатель выполнимости для теорий (SMT-решатель), который находит конкретную конфигурацию операций и связей, удовлетворяющую всем заданным ограничениям [13].

Формализация задачи синтеза безцикловых программ осуществляется через постановку задачи выполнимости в рамках SMT. Необходимо найти такой набор операций и соединений между ними, чтобы при любом наборе входных данных из заданного множества программа производила правильный результат [14]. Это достигается путём введения булевской формулы или системы логических уравнений, которая включает несколько типов переменных: константы, входные переменные программы, промежуточные значения на каждом узле вычислительного графа, а также метаданные, определяющие тип операции в каждом узле и индексы источников данных для каждой операции.

Для каждой гипотетической программы формулируются логические ограничения нескольких категорий. Во-первых, задаётся операционная семантика для каждого узла графа: если узел выполняет операцию сложения, то его выходное значение должно равняться сумме входных значений, если побитовое И – то результату применения этой операции к входам, и так далее. Во-вторых, устанавливается связь между выходом последнего узла вычислительного графа и ожидаемым результатом работы программы на каждом из тестовых примеров. В-третьих, накладываются структурные ограничения на индексы источников данных: каждый узел может получать данные только от предшествующих узлов или от входных переменных, что обеспечивает ацикличность графа и корректность потока вычислений [15, 16]. Дополнительно могут вводиться ограничения на использование определённых операций или на глубину вычислительного дерева, что позволяет контролировать сложность синтезируемых программ.

Ключевым элементом современных систем синтеза является метод counterexample-guided inductive synthesis (CEGIS) – итеративный процесс, в котором синтез и верификация чередуются циклически [17]. На каждой итерации SMT-решатель генерирует программу-кандидат, которая корректна на текущем наборе примеров. Затем эта программа подвергается верификации или тестированию на расширенном множестве входных данных. Если обнаруживается контрпример – входные данные, на которых программа работает неправильно, – этот контрпример добавляется в набор ограничений, и процесс повторяется. Итерации продолжаются до тех пор, пока не будет найдена программа, корректная на всех проверенных примерах, или пока не будет доказано, что такой программы не существует в заданном пространстве поиска.

В наших экспериментах мы применяем упрощённый вариант данного подхода, предполагая наличие фиксированного и достаточно полного набора примеров вход-выход. Мы исходим из предположения, что предоставленные примеры корректно и исчерпывающе описывают желаемое поведение

программы, что позволяет избежать дополнительных итераций верификации. Вместо классического цикла CEGIS мы используем режим оптимизации, предоставляемый решателем Z3, который позволяет не только найти программу, удовлетворяющую всем примерам, но и минимизировать некоторую целевую функцию – «стоимость» программы [18, 19]. Такая стоимость может определяться различными метриками: количеством операций, глубиной вычислительного дерева, частотой использования дорогостоящих операций (например, умножения или деления) или комбинацией этих факторов. Оптимизационный подход гарантирует, что среди всех программ, удовлетворяющих заданным примерам, будет выбрана наилучшая с точки зрения выбранной метрики качества, что особенно важно для генерации эффективного кода.

Важным техническим инструментом в нашей реализации является теория битовых векторов (bit-vectors), поддерживаемая современными SMT-решателями [20, 21, 22]. Эта теория позволяет описывать вычисления над данными фиксированной битовой ширины, что точно соответствует семантике операций в реальных процессорах и языках программирования. В рамках теории битовых векторов можно формулировать ограничения на побитовые операции (И, ИЛИ, исключающее ИЛИ, отрицание, сдвиги), арифметические операции (сложение, вычитание, умножение, деление), операции сравнения и условного выбора. Все эти операции описываются точными логическими формулами, что позволяет решателю рассуждать о корректности программ на уровне битовой семантики без потери точности.

Решатель Z3, разработанный в Microsoft Research, представляет собой мощный инструмент для работы с формулами в теории битовых векторов и других теориях [23]. Он эффективно комбинирует методы решения задач выполнимости булевых формул (SAT) с методами работы в специализированных теориях (SMT), используя технику под названием DPLL(T) – расширение классического алгоритма DPLL для SAT-решателей, интегрированное с решателями для конкретных теорий. Z3 широко применяется в различных областях: верификации программного обеспечения, анализе безопасности систем, символьном выполнении программ, автоматическом доказательстве теорем и, как в нашем случае, синтезе программ [24, 25]. Благодаря продвинутым эвристикам, оптимизациям и способности работать с миллионами переменных и ограничений, Z3 позволяет автоматизированно исследовать пространство возможных компоновок операций и находить те конфигурации, которые удовлетворяют всем заданным образцам поведения.

Важным аспектом применения SMT-решателей является эффективное кодирование задачи синтеза. Наивное кодирование может привести к экспоненциальному росту размера формулы или к созданию ограничений, которые сложно решить на практике. Поэтому используются различные техники оптимизации: симметрия-устраняющие ограничения (symmetry-breaking constraints) для исключения эквивалентных решений, инкрементальное добавление ограничений, использование теорий более высокого уровня вместо редукции к булевым формулам [26, 27]. Все эти методы в совокупности делают возможным синтез программ разумной сложности за приемлемое время, открывая путь к практическому применению автоматического синтеза в разработке программного обеспечения.

III. РЕАЛИЗАЦИЯ СИНТЕЗАТОРА ПРОГРАММ

Синтезатор реализован на языке C# с использованием библиотеки Microsoft.Z3, которая предоставляет высокоуровневый API для работы с SMT-решателем Z3. Идея состоит в том, чтобы закодировать структуру программы и её семантику в виде оптимизируемой задачи Z3. Для этого задаются следующие наборы переменных и ограничений.

Синтезируемая программа представляется в виде ациклического графа вычислений, где каждый узел соответствует промежуточному результату применения одной операции. Количество узлов задаётся параметром конфигурации NodeCount. Каждому узлу сопоставляются переменные типа IntExpr в терминологии Z3: код операции и несколько индексов для выбора операндов. Поддерживаемые операции представлены перечислением Op и включают операцию Pass (пробрасывание значения без изменений), Const (использование константы), Input (использование входной переменной), унарную операцию Not (побитовое отрицание), а также бинарные операции: битовые And, Or, Xor; арифметические Add, Sub, Mul, Div, Mod; битовые сдвиги Shl и Shr.

Для каждого узла создаются следующие переменные: _operations[nodeIndex] хранит код операции, _leftIndices[nodeIndex] и _rightIndices[nodeIndex] указывают на источники левого и правого операндов соответственно, _constantIndices[nodeIndex] определяет, какая из допустимых констант будет использована, если выбрана операция Const, и _inputIndices[nodeIndex] указывает на входную

переменную при выборе операции Input. Эти индексы указывают либо на один из ранее вычисленных узлов (для обеспечения прямой зависимости в графе вычислений), либо на один из входных параметров программы.

Ограничения на операции добавляются в методе AddOperationConstraints: код операции должен находиться в диапазоне от 0 до максимального значения перечисления (в данном случае до Op.Shr). Это гарантирует, что решатель не выберет недопустимое значение операции.

Ограничения на индексы источников реализованы в методе AddIndexConstraints. Левый и правый индексы должны указывать либо на входные переменные (индексы от 0 до InputCount-1), либо на результаты предыдущих узлов (индексы от InputCount до InputCount + nodeIndex - 1). Таким образом, максимальный допустимый индекс для узла с номером nodeIndex равен InputCount + nodeIndex - 1. Это ограничение гарантирует ацикличность графа вычислений, так как каждый узел может ссылаться только на входы или на узлы с меньшими индексами. Индекс константы ограничивается размером массива AllowedConstants, а индекс входной переменной – количеством входов программы.

Для коммутативных операций (And, Or, Xor, Add, Mul) вводятся дополнительные симметрия-устраняющие ограничения в методе AddSymmetryBreakingConstraints. Эти ограничения требуют, чтобы правый индекс был не больше левого. Это сокращает число эквивалентных вариаций программы и существенно ускоряет поиск. Например, выражения $(x + y)$ и $(y + x)$ семантически эквивалентны, но без такого ограничения решатель рассматривал бы их как два разных варианта. Условие применяется через импликацию: если операция узла является коммутативной, то должно выполняться ограничение на индексы. Это реализуется с помощью конструкции MkImplies, которая добавляет условное ограничение в оптимизатор.

Наиболее сложная часть синтезатора – это кодирование семантики операций. Для каждого узла и каждого примера вход-выход задаются булевы условия корректности в методе AddSemanticConstraints. Мы создаём переменную битового вектора _nodeValues[nodeIndex, exampleIndex], которая представляет значение данного узла при выполнении на конкретном примере. Размер битового вектора задаётся параметром BitVectorSize в конфигурации.

Для вычисления значения узла строится сложное выражение в методе BuildNodeExpression. Это выражение использует условные конструкции ITE (if-then-else) для выбора правильной операции в зависимости от значения переменной _operations[nodeIndex]. Структура выражения представляет собой каскад вложенных условий: если операция равна Const, возвращается константа; иначе, если операция равна Input, возвращается входная переменная; иначе, если операция равна Pass, возвращается значение левого источника без изменений; и так далее для каждой поддерживаемой операции.

Для получения значений операндов используются вспомогательные функции GetSourceValue, GetConstantValue и GetInputValue. Эти функции принимают индекс (который сам является переменной решателя) и возвращают соответствующее значение. Поскольку индекс неизвестен на этапе построения ограничений, используется техника построения условного дерева через BuildBinarySearchITE. Этот метод рекурсивно создаёт сбалансированное дерево условных выражений ITE, которое выбирает нужный элемент из массива по индексу. Использование двоичного поиска вместо линейной цепочки условий значительно уменьшает глубину вложенности выражений, что критично для производительности решателя. Например, для выбора одного из 8 элементов линейная схема потребует 7 вложенных ITE, тогда как двоичное дерево - только 3 уровня.

После построения выражения для узла добавляется ограничение равенства _nodeValues[nodeIndex, exampleIndex] = nodeExpression, которое говорит решателю, что значение переменной узла должно равняться результату вычисления соответствующего выражения. Для реализации операций используются специализированные функции Z3 для битовых векторов: MkBVNot для побитового отрицания, MkBVAND, MkBVOR, MkBVXOR для логических операций, MkBVAdd, MkBVSub, MkBVMul, MkBVSDiv, MkBVSMod для арифметических операций (с поддержкой знаковой арифметики), и MkBVSHL, MkBVASHR для битовых сдвигов влево и арифметического сдвига вправо соответственно.

Метод AddOutputConstraints добавляет критически важное ограничение: значение последнего узла (с индексом NodeCount - 1) должно равняться ожидаемому выходу для каждого примера. Это связывает синтезируемую программу с целевой спецификацией. Без этих ограничений решатель мог бы найти произвольную корректную программу, не обязательно удовлетворяющую примерам.

Для того, чтобы инструмент выбирал «лучший» найденный вариант, формируется арифметическое выражение – суммарная стоимость программы в методе CalculateTotalCost. Каждой операции приписана

стоимость: операция Pass имеет нулевую стоимость (так как она не выполняет никаких вычислений), операции Const и Input имеют стоимость 1, унарная операция Not – стоимость 2, а все бинарные операции (And, Or, Xor, Add, Sub, Mul, Div, Mod, Shl, Shr) имеют стоимость 3. Стоимость каждого узла вычисляется через вложенные условные выражения ITE, и все стоимости суммируются. Затем оптимизатор Z3 минимизирует это выражение с помощью вызова optimizer.MkMinimize(totalCost). В результате решения задача получения модели превращается не просто в вопрос выполнимости (SAT), а в задачу оптимизации. Это обеспечивает синтез компактных программ, так как среди всех корректных решений будет выбрано то, которое имеет минимальную стоимость.

После формирования всех ограничений решатель запускается методом optimizer.Check(). Если возвращается статус SATISFIABLE, то существует программа, удовлетворяющая всем ограничениям. Из модели решателя извлекаются конкретные значения переменных операций и индексов с помощью model.Eval(). Класс ProgramReconstructor отвечает за преобразование этих значений в читаемое строковое представление. Метод Reconstruct рекурсивно строит строковое представление программы: в соответствии с моделью преобразует каждый узел к строке, подставляя выражения для левого и правого аргументов через метод NodeToString. При этом учитывается семантика каждой операции: для бинарных операций вставляется соответствующий символ (например, &, |, ^, +, -), для унарных добавляется префикс (например, ~), а для Pass просто подставляется источник. Метод SourceToString преобразует индекс источника либо в имя входной переменной (если индекс указывает на вход), либо рекурсивно вызывает NodeToString для предыдущего узла.

Таким образом получается финальное арифметико-логическое выражение. Например, целевая функция $f(x, y) = (x + y)(x - y)$ была синтезирована как $\big((y + x)(x - y)\big)$, что семантически эквивалентно исходному, но с перестановкой слагаемых благодаря ограничениям симметрии.

Важной особенностью метода является то, что он учитывает семантику операций на всем пути построения выражения. Это позволяет SMT-синтезатору обнаруживать неожиданные эквивалентности: например, функция $(x \oplus y) + (x \wedge y)$ была сокращена до $(x \vee y)$ (см. результат опыта ниже), что классически следует из булевой алгебры и законов арифметики битовых векторов. Такой оптимизационный эффект возможен благодаря минимизации стоимости и способности Z3 выявлять эти равенства на уровне формул. Решатель использует внутренние теории битовых векторов и линейной арифметики, что позволяет ему находить неочевидные упрощения выражений, которые были бы сложны для обнаружения методами символьных преобразований или генетическими алгоритмами.

IV. ЭКСПЕРИМЕНТЫ И РЕЗУЛЬТАТЫ

Для оценки эффективности разработанного синтезатора была проведена серия экспериментов на вычислительной платформе с процессором Intel Core i5-14600KF и 32 ГБ оперативной памяти.

Все эксперименты проводились в единообразных условиях для обеспечения сопоставимости результатов. Размер битового вектора был зафиксирован на уровне 32 бита, что соответствует стандартному целочисленному типу в большинстве современных языков программирования. Набор разрешённых констант включал три базовых значения: 0, 1 и -1, представляющих нейтральные элементы основных операций и битовую маску. Выбор именно этих констант обусловлен их частым использованием в практических алгоритмах обработки данных и битовых манипуляций [28].

Каждая целевая функция верифицировалась на основе набора из четырёх пар вход-выход. Относительно небольшое количество примеров было выбрано сознательно: с одной стороны, это ускоряет процесс решения SMT-формулы, с другой стороны, четырёх примеров, как правило, достаточно для однозначной характеристики простых арифметико-логических выражений с двумя-тремя переменными. Важно отметить, что недостаточное количество примеров теоретически может привести к синтезу функции, корректной на тестовых данных, но отличающейся от целевой на других входах. Однако в рамках проведённых экспериментов такие случаи не наблюдались благодаря тщательному подбору репрезентативных тестовых наборов.

Результаты синтеза для различных целевых функций систематизированы в таблице ниже. Параметр NodeCount определяет максимальное количество вычислительных узлов в синтезируемой программе, включая финальный узел, формирующий результат. Колонка «Результат» демонстрирует полученное арифметико-логическое выражение, колонка «Стоимость» показывает суммарную стоимость программы согласно введённой метрике, а время измерялось в секундах от начала работы оптимизатора до получения решения.

Целевая функция	Node Count	Результат	Стоимость	Время, с
$f(x,y) = (x+y) * (x-y)$	3	$((y+x) * (x-y))$	9	7.2
$f(x,y) = (x+y) \gg 1$	3	$((y+x) \gg 1)$	7	6.9
$f(a,b,c) = (a*b) \oplus c$	3	$((b*a) \oplus c)$	6	6.5
$f(a,b,c) = (a \wedge b) + (\sim c)$	3	$((b \wedge a) + \sim (c))$	8	8.7
$f(x,y) = (x \oplus y) + (x \wedge y)$	3	$(x \vee y)$	3	2.9
$f(a,b,c) = (a + (b \ll 1)) \wedge c$	4	$(c \wedge (a + (b \ll 1)))$	10	226.8
$f(x_0,x_1,x_2,x_3) = (x_0+x_1) \oplus (x_2 - x_3)$	3	$((x_1+x_0) \oplus (x_2 - x_3))$	9	64.8

Анализ полученных данных выявляет несколько закономерностей. Для относительно простых функций, требующих три вычислительных узла, синтезатор демонстрирует стабильное время работы в диапазоне от 3 до 9 секунд. Это время включает построение SMT-формулы, кодирующей все структурные и семантические ограничения, процесс оптимизации целевой функции стоимости и извлечение решения из модели. Такая производительность вполне приемлема для интерактивного использования инструмента в процессе разработки.

Однако при увеличении параметра NodeCount до четырёх наблюдается большой рост времени синтеза. Наиболее показательным является пример с функцией $(a + (b \ll 1)) \wedge c$, синтез которой занял приблизительно 227 секунд. Этот скачок времени отражает фундаментальное свойство задачи синтеза программ: пространство возможных программ растёт экспоненциально с увеличением числа узлов. Каждый дополнительный узел добавляет новые переменные для выбора операции, индексов операндов и констант, что мультипликативно увеличивает сложность формулы для SMT-решателя.

Интересной особенностью является то, что синтезатор систематически применяет свойство коммутативности операций. В полученных формулах операнды коммутативных операций часто оказываются переставленными по сравнению с исходной целевой функцией: например, $(y+x)$ вместо $(x+y)$ или $(b*a)$ вместо $(a*b)$. Эта перестановка не влияет на семантическую корректность результата и является прямым следствием симметрия-устраняющих ограничений, введённых в модель. Напомним, что для коммутативных операций синтезатор принудительно упорядочивает индексы операндов согласно условию $r_i \leq l_i$, что существенно сокращает симметричное пространство поиска и ускоряет работу решателя.

Особенно примечательным является результат синтеза для функции $f(x,y) = (x \oplus y) + (x \wedge y)$. Вместо буквального воспроизведения структуры целевого выражения синтезатор вернул существенно более простую формулу $(x \vee y)$ со стоимостью всего 3, в то время как прямая реализация исходного выражения потребовала бы значительно большей стоимости.

Этот результат не является ошибкой, а демонстрирует глубокое понимание битовой арифметики, заложенное в семантические ограничения модели. Действительно, из законов булевой алгебры и свойств битовых операций следует тождество: $(x \vee y) = (x \oplus y) + (x \wedge y)$. Это можно понять интуитивно: операция XOR даёт биты, установленные ровно в одном из операндов, операция AND даёт биты, установленные в обоих операндах, а их сумма с учётом переноса эквивалентна побитовому OR, когда нет переноса между разрядами. Формально это тождество можно проверить через таблицы истинности или алгебраическое преобразование.

Данный пример иллюстрирует ключевое преимущество SMT-подхода к синтезу программ: система способна автоматически обнаруживать неочевидные алгебраические упрощения без явного программирования правил оптимизации. Механизм минимизации целевой функции стоимости в сочетании с полной семантической моделью битовых операций естественным образом ведёт решатель к математически эквивалентным, но более компактным выражениям. Фактически, синтезатор выполняет роль супероптимизатора, находящего глобальный оптимум в пространстве семантически эквивалентных программ заданного размера.

Важным свойством предложенного подхода является формальная гарантия корректности. Поскольку синтезатор строит полную модель семантики функции через ограничения на значения узлов для всех заданных примеров, любая найденная программа по построению корректна на этих примерах. При достаточном представительном наборе тестовых случаев это обеспечивает высокую вероятность корректности на всём пространстве входов, особенно для небольших детерминированных функций.

Минимизация метрики стоимости гарантирует получение кратчайшей программы в заданной модели вычислений. Это свойство превращает синтезатор в инструмент супероптимизации для небольших фрагментов кода, автоматически находящий наиболее эффективную реализацию заданной

функциональности [29]. В контексте компиляторных оптимизаций или оптимизации критических по производительности участков кода такая возможность представляет значительную практическую ценность.

Тем не менее необходимо учитывать фундаментальные ограничения масштабируемости подхода. Сложность задачи синтеза растёт экспоненциально как с увеличением количества узлов NodeCount, так и с ростом числа входных переменных и обучающих примеров. Каждый дополнительный узел вводит новые целочисленные переменные для выбора операции и индексов, а каждый дополнительный пример добавляет новые битовые переменные для значений узлов и соответствующие ограничения. Современные SMT-решатели, несмотря на впечатляющие достижения последних лет, всё ещё испытывают трудности с формулами, содержащими тысячи переменных и сложные нелинейные зависимости.

Для синтеза более сложных функций, требующих большего количества узлов или оперирующих с большим числом переменных, могут потребоваться дополнительные техники. Перспективными направлениями являются декомпозиционные подходы, разбивающие сложную функцию на композицию более простых компонентов, иерархический синтез с промежуточными абстракциями, а также гибридные методы, комбинирующие SMT-синтез с эвристическим поиском или машинным обучением для предсказания перспективных структур программ [30].

ЗАКЛЮЧЕНИЕ

Разработанный подход демонстрирует, что SMT-решатель Z3 можно эффективно использовать для синтеза простых арифметико-логических выражений по образцам. Предложенная реализация автоматически подбирает последовательность операций и их связей так, чтобы удовлетворять всем вход-выходным примерам, и при этом минимизирует суммарную «стоимость» программы. Это позволяет получать более компактные и оптимальные по метрике реализации функций (как, например, редукция $(x \oplus y) + (x \wedge y)$ в $x \vee y$). Подход обеспечивает корректность решений по построению, что важно для приложений верификации и автоматической оптимизации программ.

БИБЛИОГРАФИЯ

[1] Massalin H. Superoptimizer: a look at the smallest program // ACM SIGARCH Computer Architecture News. IEEE Computer Society Press. – 1987. – Vol. 15. – №. 5. – Pp. 122–126. DOI: 10.1145/36206.36194.

[2] Souper: A synthesizing superoptimizer / Sasnauskas R., Chen Y., Collingbourne P. [et al.] // arXiv preprint arXiv: 1711.04422. – 2017. – Pp. 1-14. – DOI: 10.48550/arXiv.1711.04422.

[3] Scaling up superoptimization / Phothilimthana P. M., Thakur A., Bodik R. [et al.] // ACM SIGARCH Computer Architecture News. ACM. – 2016. – Vol. 51. – №. 4. – Pp. 297–310. – DOI: 10.1145/2954679.2872387.

[4] Syntax-guided synthesis / Alur R., Bodik R., Juniwalet G. [et al.] // Formal Methods in Computer-Aided Design. IEEE. – 2013. – Pp. 1-8. – DOI: 10.1109/FMCAD.2013.6679385.

[5] Synthesis of loop-free programs / Gulwani S., Jha S., Tiwari A. [et al.] // PLDI. 2011. Vol. 11. P. 62–73. – DOI: 10.1145/1993498.1993506.

[6] Beyer D., Dangl M., Wendler P. A unifying view on SMT-based software verification // Journal of Automated Reasoning. – 2018. – Vol. 60. – № 3. – Pp. 299-335. – DOI: 10.1007/s10817-017-9432-6.

[7] Component based synthesis applied to bitvector programs: Technical Report MSR-TR-2010-12 / Gulwani S., Jha S., Tiwari A. [et al.] / Microsoft Research. – 2011. – 14 p.

[8] De Moura L., Bjørner N. Z3: An efficient SMT solver // Tools and Algorithms for the Construction and Analysis of Systems (TACAS). – Berlin; Heidelberg: Springer, 2008. – Vol. 4963. – Pp. 337-340. – DOI: 10.1007/978-3-540-78800-3_24.

[9] Gulwani S. Dimensions in program synthesis // Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming. – 2010. – Pp. 13–24. – DOI: 10.1145/1836089.1836091.

[10] Monniaux D. A survey of satisfiability modulo theories // Computer Engineering & Science. 2016. – Vol. 9890. – Pp. 401-425. – DOI: 10.1007/978-3-319-45641-6_26.

[11] Андрианов И. А., Григорьева А. Н. Модернизация индекса для поиска по регулярным выражениям // Системы управления и информационные технологии. – 2020. – № 2 (80). – С. 60-64.

[12] Козлов С. В. Интерпретация инвариантов теории графов в контексте применения соответствия Галуа при создании и сопровождении информационных систем // International Journal of Open Information Technologies. – 2016. – Т. 4. № 7. – С. 38-44.

[13] What's decidable about syntax-guided synthesis? / Caulfield T., Alur R., Fisman D. [et al.] // arXiv preprint arXiv:1510.08393. – 2015. – Pp. 1-15. – DOI: 10.48550/arXiv.1510.08393.

[14] Functional synthesis for linear arithmetic and sets / Kuncak V., Mayer M., Piskac R. [et al.] // International Journal on Software Tools for Technology Transfer. Springer. – 2013. – Vol. 15 № 5-6. – Pp. 455-474. – DOI: 10.1007/s10009-011-0217-7.

[15] Solving quantified bit-vectors using invertibility conditions / Niemetz A., Preiner S., Krebbers R. [et al.] // International Conference on Computer Aided Verification. Springer. 2018. – Pp. 236-255. – DOI: 10.1007/978-3-319-96142-2_16.

[16] Советов П. Н. Синтез линейных программ для стековой машины // Высокопроизводительные вычислительные системы и технологии. – 2019. – Т. 3, № 1. – С. 17-22.

[17] Solar-Lezama A. Program synthesis by sketching // Ph.D. Dissertation. – Berkeley. University of California. – 2008. – 176 p.

[18] Synthesising programs with non-trivial constants / Abate A., Barbosa H., Barrett C. [et al.] // Journal of automated reasoning. – 2023. – Vol. 67. – №2. – P. 19. – DOI: 10.1007/s10817-023-09664-4.

[19] Reynolds A. Challenges for fast synthesis procedures in SMT // ARCADE 2017. 1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements. – 2017. – Vol. 51. – Pp. 69-75. – DOI: 10.29007/xrhz.

[20] Abraham E., Kremer G. SMT solving for arithmetic theories: Theory and tool support // International Conference on Symbolic

and Numeric Algorithms for Scientific Computing. IEEE. – 2017. – Pp. 1-8. – DOI: 10.1109/SYNASC.2017.00009.

[21] Reynolds A., King T., Kuncak V. Solving quantified linear arithmetic by counterexample-guided instantiation // Formal Methods in System Design. – 2017. – Vol. 51. – Pp. 1-33. DOI: 10.1007/s10703-017-0290-y.

[22] Spielmann A., Kuncak V. Synthesis for unbounded bit-vector arithmetic // International Joint Conference on Automated Reasoning. – 2012. – Pp. 499-513. – DOI: 10.1007/978-3-642-31365-3_39.

[23] Barbosa H. An introduction to SMT solving with quantifiers // Universidade Federal de Minas Gerais. – 2024. – 111 p.

[24] Козлов С. В., Светлаков А. В. Применение регулярных выражений для обработки текстовых данных // International Journal of Open Information Technologies. – 2022. – Т. 10, № 9. – С. 82-89.

[25] Козлов С. В., Светлаков А. В. О LL(1)-грамматиках, алгоритмах на них и методах их анализа в программировании // International Journal of Open Information Technologies. – 2022. – Т. 10, № 3. – С. 30-38.

[26] Boosting SMT solver performance on mixed-bitwise-arithmetic expressions / Xu D., Liu B., Feng W. [et al.] // Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. – 2021. – Pp. 651–664. – DOI:10.1145/3453483.3454068.

[27] Schindler T. SMT solving, interpolation, and quantifiers // Ph.D. Dissertation. – University of Freiburg. – 2022. – 229 p. – DOI: 10.6094/UNIFR/229572.

[28] Антипина А. В. Задача автоматической генерации peephole-оптимизаций: обзор подходов, решение проблемы оптимального расширения архитектуры набора managed инструкций // Современные информационные технологии и ИТ-образование. – 2021. – Т. 17, № 3. С. 613-624. – DOI 10.25559/SITITO.17.202103.613-624.

[29] Гращенков Н. Р., Козлов С. В. Создание восьмибитного сумматора в видеоигре "Minecraft" / Н. Р. Гращенков, // Новые информационные технологии и системы (НИТиС-2023): сборник научных статей по материалам XX Международной научно-технической конференции, посвященной 80-летнему юбилею Пензенского государственного университета, Пенза, 16–17 ноября 2023 года. – Пенза: Пензенский государственный университет, 2023. – С. 174-178.

[30] Balunovic M., Bielik P., Vechev M. Learning to solve SMT formulas // Proceedings of the 32nd International Conference on Neural Information Processing Systems. – 2018. – Pp. 10338-10349.

Synthesis of Arithmetic-Logical Expressions Using an SMT Solver

S. V. Kozlov, M. D. Cherepanov

Abstract – The paper investigates a method for automatic synthesis of arithmetic-logical programs based on the Z3 SMT solver. The Syntax-Guided Synthesis (SyGuS) approach is considered in relation to loop-free programs represented as acyclic computation graphs. The problem of synthesis is formalized through a system of logical constraints in the theory of bit-vectors, which includes the operational semantics of nodes, structural constraints on data flow, and symmetry-breaking conditions for commutative operations. A practical implementation of the synthesizer in C# using the Microsoft.Z3 library is described; it encodes the structure and semantics of the program as an optimization problem with cost function minimization. The experimental section demonstrates the synthesis of various target functions with two to four input variables: runtime ranges from 2.9 to 227 seconds depending on the expression's complexity. It is shown that the synthesizer can automatically detect non-obvious algebraic equivalences and generate cost-optimal implementations. Issues of scalability and the exponential growth of complexity with an increasing number of computational nodes are discussed. The results confirm the applicability of SMT-based technologies to the super optimization of small program fragments with formal correctness guarantees.

Keywords – program synthesis, formal verification, Satisfiability Modulo Theories, SMT solver, Z3, Syntax-Guided Synthesis, bit-vectors, Boolean-arithmetic expressions, automatic program optimization, programming.

REFERENCES

[1] Massalin H. Superoptimizer: a look at the smallest program // ACM SIGARCH Computer Architecture News. IEEE Computer Society Press. – 1987. – Vol. 15. – #. 5. – Pp. 122–126. DOI: 10.1145/36206.36194.

[2] Souper: A synthesizing superoptimizer / Sasnauskas R., Chen Y., Collingbourne P. [et al.] // arXiv preprint arXiv: 1711.04422. – 2017. – Pp. 1-14. – DOI: 10.48550/arXiv.1711.04422.

[3] Scaling up superoptimization / Phothilimthana P. M., Thakur A., Bodik R. [et al.] // ACM SIGARCH Computer Architecture News. ACM. – 2016. – Vol. 51. – #. 4. – Pp. 297–310. – DOI: 10.1145/2954679.2872387.

[4] Syntax-guided synthesis / Alur R., Bodik R., Juniwalet G. [et al.] // Formal Methods in Computer-Aided Design. IEEE. – 2013. – Pp. 1-8. – DOI: 10.1109/FMCAD.2013.6679385.

[5] Synthesis of loop-free programs / Gulwani S., Jha S., Tiwari A. [et al.] // PLDI. 2011. Vol. 11. P. 62–73. – DOI: 10.1145/1993498.1993506.

[6] Beyer D., Dangl M., Wendler P. A unifying view on SMT-based software verification // Journal of Automated Reasoning. – 2018. – Vol. 60. – # 3. – Pp. 299-335. – DOI: 10.1007/s10817-017-9432-6.

[7] Component based synthesis applied to bitvector programs: Technical Report MSR-TR-2010-12 / Gulwani S., Jha S., Tiwari A. [et al.] / Microsoft Research. – 2011. – 14 p.

[8] De Moura L., Bjørner N. Z3: An efficient SMT solver // Tools and Algorithms for the Construction and Analysis of Systems (TACAS). – Berlin; Heidelberg: Springer, 2008. – Vol. 4963. – Pp. 337-340. – DOI: 10.1007/978-3-540-78800-3_24.

[9] Gulwani S. Dimensions in program synthesis // Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming. – 2010. – Pp. 13–24. – DOI: 10.1145/1836089.1836091.

[10] Monniaux D. A survey of satisfiability modulo theories // Computer Engineering & Science. 2016. – Vol. 9890. – Pp. 401-425. – DOI: 10.1007/978-3-319-45641-6_26.

[11] Andrianov I. A., Grigor'eva A. N. Modernizacija indeksa dlja poiska po reguljarnym vyrazhenijam // Sistemy upravlenija i informacionnye tehnologii. – 2020. – # 2 (80). – S. 60-64.

[12] Kozlov S. V. Interpretacija invariantov teorii grafov v kontekste primenenija sootvetstvija Galua pri sozdanii i soprovozhdenii informacionnyh sistem // International Journal of Open Information Technologies. – 2016. – T. 4. # 7. – S. 38-44.

[13] What's decidable about syntax-guided synthesis? / Caulfield T., Alur R., Fisman D. [et al.] // arXiv preprint arXiv:1510.08393. – 2015. – Pp. 1-15. – DOI: 10.48550/arXiv.1510.08393.

[14] Functional synthesis for linear arithmetic and sets / Kuncak V., Mayer M., Piskac R. [et al.] // International Journal on Software Tools for Technology Transfer. Springer. – 2013. – Vol. 15 # 5-6. – Pp. 455-474. – DOI: 10.1007/s10009-011-0217-7.

[15] Solving quantified bit-vectors using invertibility conditions / Niemetz A., Preiner S., Krebbers R. [et al.] // International Conference on Computer Aided Verification. Springer. 2018. – Pp. 236-255. – DOI: 10.1007/978-3-319-96142-2_16.

[16] Sovetov P. N. Sintez linejnyh programm dlja stekovoj mashiny // Vysokoproizvoditel'nye vychislitel'nye sistemy i tehnologii. – 2019. – T. 3, # 1. – S. 17-22.

[17] Solar-Lezama A. Program synthesis by sketching // Ph.D. Dissertation. – Berkeley. University of California. – 2008. – 176 p.

[18] Synthesising programs with non-trivial constants / Abate A., Barbosa H., Barrett C. [et al.] // Journal of automated reasoning. – 2023. – Vol. 67. – #2. – P. 19. – DOI: 10.1007/s10817-023-09664-4.

[19] Reynolds A. Challenges for fast synthesis procedures in SMT // ARCADE 2017. 1st International Workshop on Automated Reasoning: Challenges,

Applications, Directions, Exemplary Achievements. – 2017. – Vol. 51. – Pp. 69-75. – DOI: 10.29007/xrhz.
[20] Abraham E., Kremer G. SMT solving for arithmetic theories: Theory and tool support // International Conference on Symbolic and Numeric Algorithms for Scientific Computing. IEEE. – 2017. – Pp. 1-8. – DOI: 10.1109/SYNASC.2017.00009.
[21] Reynolds A., King T., Kuncak V. Solving quantified linear arithmetic by counterexample-guided instantiation // Formal Methods in System Design. – 2017. – Vol. 51. – Pp. 1-33. DOI: 10.1007/s10703-017-0290-y.
[22] Spielmann A., Kuncak V. Synthesis for unbounded bit-vector arithmetic // International Joint Conference on Automated Reasoning. – 2012. – Pp. 499-513. – DOI: 10.1007/978-3-642-31365-3_39.
[23] Barbosa H. An introduction to SMT solving with quantifiers // Universidade Federal de Minas Gerais. – 2024. – 111 p.
[24] Kozlov S. V., Svetlakov A. V. Primenenie reguljarnyh vyrazhenij dlja obrabotki tekstovyh dannyh // International Journal of Open Information Technologies. – 2022. – T. 10, # 9. – S. 82-89.
[25] Kozlov S. V., Svetlakov A. V. O LL(1)-grammatikah, algoritmah na nih i metodah ih analiza v programmirovanii // International Journal of Open Information Technologies. – 2022. – T. 10, # 3. – S. 30-38.
[26] Boosting SMT solver performance on mixed-bitwise-arithmetic expressions / Xu D., Liu B., Feng W. [et al.] // Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. – 2021. – Pp. 651–664. – DOI:10.1145/3453483.3454068.
[27] Schindler T. SMT solving, interpolation, and quantifiers // Ph.D. Dissertation. – University of Freiburg. – 2022. – 229 p. – DOI: 10.6094/UNIFR/229572.
[28] Antipina A. V. Zadacha avtomaticheskoj generacii peephole-optimizacij: obzor podhodov, reshenie problemy optimal'nogo rasshirenija arhitektury nabora managed instrukcij // Sovremennye informacionnye tehnologii i IT-obrazovanie. – 2021. – T. 17, # 3. S. 613-624. – DOI 10.25559/SITITO.17.202103.613-624.
[29] Grashhenkov N. R., Kozlov S. V. Sozdanie vos'mibitnogo summatora v videoigre "Minecraft" / N. R. Grashhenkov, // Novye informacionnye tehnologii i sistemy (NITiS-2023): sbornik nauchnyh statej po materialam XX Mezhdunarodnoj nauchno-tehnicheskoj konferencii, posvjashhennoj 80-letnemu jubileju Penzenskogo gosudarstvennogo universiteta, Penza, 16–17 nojabrja 2023 goda. – Penza: Penzenskij gosudarstvennyj universitet, 2023. – S. 174-178.
[30] Balunovic M., Bielik P., Vechev M. Learning to solve SMT formulas // Proceedings of the 32nd International Conference on Neural Information Processing Systems. – 2018. – Pp. 10338-10349.