МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
КАФЕДРА АМ

Исследовательская работа

Тема: Применение SAT-решателей для верификации комбинационных схем

Студент гр. 3312 _____ Барченков П.А.

Студент гр. 3312 _____ Лебедев И.А.

Студент гр. 3312 _____ Шарапов И.Д.

Преподаватель _____ Поздняков С.Н.

Санкт-Петербург
2025

Содержание

ЦЕЛЬ ИССЛЕДОВАНИЯ 4

ЗАДАНИЕ 4

ВВЕДЕНИЕ 5

1. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ВЕРИФИКАЦИИ КОМБИНАЦИОННЫХ СХЕМ И ИСПОЛЬЗОВАНИЯ SAT-РЕШАТЕЛЕЙ 7

1.1. Комбинационные логические схемы и булевы функции 7

1.1.1. Базис логических элементов 7

1.1.2. Представление схем в виде булевых формул 8

1.1.3. Связь между схемой и булевой функцией 10

1.2. Постановка задачи проверки эквивалентности схем 10

1.3. Задача SAT. Метод митора для проверки эквивалентности 11

1.4. Алгоритмы SAT-решения: DPLL, CDCL. Особенности Z3 13

1.4.1. Классический алгоритм DPLL 14

1.4.2. Современные CDCL-решатели 15

1.4.3. Особенности решателя Z3 и его использование в работе 18

1.5. Выводы по первой главе 19

2. РАЗРАБОТКА ПРОГРАММНОГО МОДУЛЯ ПРОВЕРКИ ЭКВИВАЛЕНТНОСТИ СХЕМ 22

2.1. Постановка требований к программному модулю и выбор инструментальных средств 22

2.2. Формат представления входных данных и модуль разбора булевых формул 24

2.2.1. Синтаксис булевых формул 25

2.2.2. Лексический анализ (функция tokenize) 25

2.2.3. Структура абстрактного синтаксического дерева 26
2.2.4. Синтаксический анализ 27
2.3. Реализация традиционного метода проверки эквивалентности (полный перебор) 28
2.4. Реализация SAT-подхода 30
2.4.1. Проверка эквивалентности с использованием решателя Z3 30
2.4.2. Учебная реализация DPLL (митор → КНФ → DPLL) 32
2.4.3. Учебная реализация CDCL (митор → КНФ → CDCL) 34
2.5. Структура программного комплекса и вспомогательные модули 35
2.6. Выводы по второй главе 36
3. ЭКСПЕРИМЕНТАЛЬНОЕ ИССЛЕДОВАНИЕ ЭФФЕКТИВНОСТИ МЕТОДОВ 38
3.1. Постановка эксперимента 38
3.2. Описание программных модулей 38
3.3. Ручные эксперименты 39
3.4. Автоматические эксперименты 40
3.5. Выводы по третьей главе 42
ЗАКЛЮЧЕНИЕ 43
СПИСОК ЛИТЕРАТУРЫ 45

ЦЕЛЬ ИССЛЕДОВАНИЯ

Реализовать верификацию схемы с использованием Z3 или MiniSat и сравнить с традиционными методами.

ЗАДАНИЕ

1. Проанализировать теоретические основы задачи выполнимости булевых формул и алгоритмов SAT-решения, а также формальную постановку задачи проверки эквивалентности комбинационных схем.
2. Разработать способ представления комбинационных схем и построения митора, сводящего проверку эквивалентности к проверке выполнимости одной булевой формулы.
3. Реализовать программный модуль традиционной проверки эквивалентности на основе полного перебора входных векторов.
4. Реализовать программный модуль проверки эквивалентности схем с использованием SAT-решателя Z3.
5. Провести экспериментальное сравнение времени работы традиционного метода и SAT-подхода на наборе тестовых схем различной размерности и сложности.
6. Проанализировать полученные экспериментальные результаты и сформулировать выводы о применимости SAT-решателей для верификации комбинационных схем.

Объектом исследования в работе являются комбинационные логические схемы, представимые в виде булевых функций от конечного числа входных переменных.

Предметом исследования является методика проверки эквивалентности таких схем с использованием SAT-решателей и её сравнение с традиционным методом полного перебора.

ВВЕДЕНИЕ

Развитие современной микроэлектроники и средств автоматизированного проектирования (EDA-систем) привело к существенному росту сложности цифровых устройств. Современные интегральные схемы содержат миллионы и миллиарды логических элементов, а проектирование даже относительно небольших блоков всё чаще осуществляется с использованием языков описания аппаратуры и автоматических средств синтеза и оптимизации. В этих условиях задача верификации корректности проектируемых схем приобретает ключевое значение.

Одной из базовых задач верификации является проверка эквивалентности комбинационных схем. На практике она возникает, например, при сравнении:

- эталонного описания схемы и результата логического синтеза;
- схемы до и после оптимизации (упрощения логики, удаления избыточных элементов);
- двух различных реализаций одной и той же функциональности.

Под эквивалентностью обычно понимается совпадение значений всех выходов схем для всех возможных наборов входных сигналов. Традиционный, «лобовой» подход к решению этой задачи основан на полном переборе всех входных векторов: для каждого набора входных переменных схемы моделируются, и их выходы сравниваются. Однако число возможных входных комбинаций растёт экспоненциально от количества входов n и составляет 2^n. Уже при нескольких десятках входов полный перебор становится практически неприменим из-за недопустимого времени вычислений.

Альтернативой традиционному перебору является подход, основанный на сведении задачи эквивалентности к задаче выполнимости булевых формул (SAT). Идея состоит в том, чтобы представить поведение сравниваемых комбинационных схем в виде булевых функций от общих входных

переменных, построить так называемый митор – схему или формулу, которая равна единице тогда и только тогда, когда выходы исходных схем различаются, и затем передать полученную формулу на вход SAT-решателю. Если SAT-решатель обнаруживает набор входов, при котором митор равен единице (формула выполнима), то схемы неэквивалентны, и этот набор входов служит контрпримером. Если же формула оказывается невыполнимой, это означает, что для любых входов выходы схем совпадают, то есть схемы эквивалентны.

Современные SAT-решатели (такие как Z3 и MiniSat) реализуют эффективные алгоритмы на основе DPLL/CDCL, включающие развёрнутые механизмы предобработки формулы, эвристики выбора переменных, обучение конфликтам и др. Благодаря этому они способны обрабатывать формулы с сотнями тысяч переменных и миллионами клауз, что делает подход на основе SAT перспективным для задач верификации цифровых схем. При этом по сравнению с полным перебором SAT-подход не требует явно проходить по всем входным векторам и зачастую демонстрирует существенно лучшую масштабируемость.

В данной работе рассматривается применение SAT-решателей к задаче проверки эквивалентности комбинационных схем. В качестве традиционного метода используется моделирование поведения схем методом полного перебора по входным переменным, а в качестве SAT-подхода – построение митора и передача соответствующей булевой формулы в решатель Z3. Для сопоставления методов разрабатывается программный комплекс, реализующий оба подхода, и проводится экспериментальное сравнение их временной эффективности на наборе тестовых примеров.

1. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ВЕРИФИКАЦИИ КОМБИНАЦИОННЫХ СХЕМ И ИСПОЛЬЗОВАНИЯ SAT-РЕШАТЕЛЕЙ

1.1. Комбинационные логические схемы и булевы функции

Комбинационная логическая схема – это устройство, в котором значения выходных сигналов в каждый момент времени однозначно определяются текущими значениями входов и не зависят от истории работы схемы. Формально, если схема имеет n входов и m выходов, она реализует отображение

$$f: \{0,1\}^n \to \{0,1\}^m.$$

Здесь $\{0,1\}$ обычно интерпретируется как множество логических значений: 0 – «ложь»/низкий уровень, 1 – «истина»/высокий уровень.

Каждый выходной сигнал можно рассматривать как отдельную булеву функцию:

$$f(x_1,\dots,x_n) = \big(f_1(x_1,\dots,x_n),\dots,f_m(x_1,\dots,x_n)\big),$$

где $f_i: \{0,1\}^n \to \{0,1\}$ – булева функция, задающая значение i-го выхода при заданном наборе входных переменных $(x_1,\dots,x_n)$.

1.1.1. Базис логических элементов

На уровне схем логика задаётся сетью элементарных логических элементов. Наиболее распространённый базис включает:

- элемент И (AND): $z = x \wedge y$;
- элемент ИЛИ (OR): $z = x \vee y$;
- элемент НЕ (NOT): $z = \neg x$.

Кроме того, часто используются:

- исключающее ИЛИ (XOR):$z = x \oplus$ y;
- импликация: $z = x \to y$
- эквивалентность: $z = x \leftrightarrow y$

Известно, что базиса {∧,∨, ¬} достаточно для реализации любой булевой функции. Это означает, что любую комбинационную схему можно представить как формулу булевой алгебры над переменными $x_1, \ldots, x_n$ и операциями ∧, ∨, ¬, а при необходимости – и другими производными операциями.

1.1.2. Представление схем в виде булевых формул

Использование булевых формул даёт удобный абстрактный уровень. Вместо явного описания логической схемы (графа) можно работать с выражениями вида:

$$f(x_1, \ldots, x_n) = (x_0 \wedge x_1) \vee \big(\neg x_2 \wedge (x_1 \vee x_3)\big),$$

или, в текстовом виде, как в реализованной программе:

```
(x0 & x1) | (!x2 & (x1 | x3))
```

В таком подходе:

- каждая переменная `x0, x1, x2, …` соответствует входному сигналу схемы;
- булево выражение задаёт функциональную зависимость выхода от этих входов;
- одна формула = одна выходная комбинационная схема.

В случае многовыходной схемы каждый выход может описываться своей формулой:

```
y0 = (x0 & x1) | x2
y1 = (!x0 & x3) | (x1 & x2)
...
```

В программной реализации, использующей SAT-решатель Z3, такие формулы:

- задаются в виде строк;
- разбираются парсером в абстрактное синтаксическое дерево (AST);
- далее либо вычисляются напрямую (для брутфорса), либо переводятся в выражения Z3.

Булевы выражения Z3 (далее “форма Z3”). Z3 – промышленный SMT/SAT-решатель, предоставляющий API для построения логических ограничений и проверки их выполнимости. В рамках данной работы используется лишь пропозициональная (булева) часть Z3: переменные имеют тип Bool, а формулы строятся с помощью логических связок And, Or, Not и производных конструкций. Под “формой Z3” далее понимается объект Z3-выражения (AST Z3), представляющий булеву формулу, которую можно передать в Solver для проверки sat/unsat.

Пример. Форма Z3 для формулы $F = !(x0\&x1) \mid x2$

```
x0 = Bool("x0")
x1 = Bool("x1")
x2 = Bool("x2")
Fz3 = Or(Not(And(x0, x1)), x2)
```

Fz3 — это “форма Z3”: готовое выражение, которое можно добавить в Solver().

Абстрактное синтаксическое дерево (AST). Для обработки входного выражения вида !(x0 & x1) | x2 сначала выполняется синтаксический разбор. Результатом разбора является абстрактное синтаксическое дерево (AST, Abstract Syntax Tree) — дерево, в узлах которого находятся операции (!, &, |, ->, <->), а в листьях — переменные (x0, x1, …). AST отражает структуру формулы (что с чем связано) и позволяет далее единообразно: (1) вычислять значение формулы при заданном наборе входов, (2) преобразовывать формулу в представление Z3, (3) строить митор.

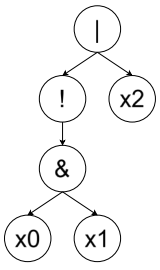

Рисунок 1 – AST для формулы $F = !(x0\&x1) \mid x2$

1.1.3. Связь между схемой и булевой функцией

Между схемой и булевой функцией есть прямая взаимно однозначная связь:

- любая комбинационная схема (над фиксированным набором входов) однозначно задаёт булеву функцию;
- любое выражение булевой алгебры можно реализовать комбинационной схемой в стандартном базисе.

В рамках данной работы это позволяет отказаться от явного описания элементов и считать схемой булеву формулу над переменными $x_0, x_1, \ldots, x_{n-1}$. Это упрощает программную реализацию и делает естественным использование SAT-решателя: Z3 работает именно с булевыми формулами.

1.2. Постановка задачи проверки эквивалентности схем

Пусть заданы две комбинационные схемы $C^{(1)}$ и $C^{(2)}$ с одинаковым числом входов n и выходов m. Обозначим через

$$f^{(1)}(x_1, \ldots, x_n) = \left(f_1{}^{(1)}(x), \ldots, f_m{}^{(1)}(x)\right),$$

$$f^{(2)}(x_1, \ldots, x_n) = \left(f_1{}^{(2)}(x), \ldots, f_m{}^{(2)}(x)\right)$$

соответствующие векторные булевы функции, реализуемые схемами.

Схемы называются эквивалентными, если для любого набора входных значений их выходы совпадают:

$$\forall x \in 0{,}1^n : f^{(1)}(x) = f^{(2)}(x)$$

Практически задача проверки эквивалентности возникает в следующих ситуациях:

- сравнение исходного (референсного) описания блока и результата логического синтеза;
- сравнение схемы до и после оптимизаций (удаления избыточной логики, переписывания выражений);

• сравнение двух альтернативных реализаций одной и той же спецификации.

Наивный способ решения задачи состоит в полном переборе всех входных векторов**.** Для каждого входного набора $x \in \{0,1\}^n$ вычисляются значения $f^{(1)}(x)$ и $f^{(2)}(x)$. Если хотя бы для одного входа выходы различаются, схемы признаются неэквивалентными, причём соответствующий входной набор служит контрпримером. Если же на всех 2^n входах выходы совпали, схемы эквивалентны.

Основной недостаток этого подхода – экспоненциальный рост числа проверок с увеличением количества входов. При $n = 10$ количество комбинаций составляет 1024, при $n = 20$ – уже около миллиона, а при $n = 30$ – более миллиарда. Даже при сравнительно небольших значениях n время, необходимое для полного перебора, быстро становится неприемлемым. Это мотивирует поиск методов, избегающих явного прохода по всему пространству входных векторов.

1.3. Задача SAT. Метод митора для проверки эквивалентности

Одним из распространённых подходов к проверке эквивалентности комбинационных схем является сведение этой задачи к задаче выполнимости булевых формул (SAT)**.** Задача SAT в общем виде формулируется следующим образом: по заданной булевой формуле $F(x_1, \dots, x_k)$ необходимо определить, существует ли такой набор значений переменных, при котором формула истинна:

$$\exists x \in \{0,1\}^k : F(x) = 1.$$

Для проверки эквивалентности схем $C^{(1)}$ и $C^{(2)}$ вводится вспомогательная конструкция, называемая митором (*miter*). На общие входы подаётся один и тот же вектор $\mathrm{x} \in \{0,1\}^{\mathrm{n}}$, после чего сравниваются выходы обеих схем. Пусть каждая схема имеет m выходов и задаётся вектором булевых функций

$$F^{(j)}(x) = \left(f_1^{(j)}(x), \dots, f_m^{(j)}(x)\right), \quad j \in \{1,2\}$$

Для каждого выхода i определяется булева функция

$$d_i(x) = f_i^{(1)}(x) \oplus f_i^{(2)}(x)$$

которая принимает значение 1 тогда и только тогда, когда i-е выходы двух схем различаются. Общая функция различия (многовыходный митор) определяется как дизъюнкция по всем выходам:

$$M(x) = \bigvee_{i=1}^{m} d_i(x) = \bigvee_{i=1}^{m} \left(f_i^{(1)}(x) \oplus f_i^{(2)}(x)\right).$$

Функция $M(x)$ равна 1, если хотя бы один выход схем отличается, и 0, если все выходы совпадают. Тогда эквивалентность схем можно сформулировать так:

- если $M(x)$ выполнима (SAT), существует вход x, на котором хотя бы один выход отличается; такой x является контрпримером эквивалентности, и схемы неэквивалентны;
- если $M(x)$ невыполнима (UNSAT), то для всех $x \in \{0,1\}^n$ выполняется $F^{(1)}(x) = F^{(2)}(x)$, то есть схемы эквивалентны.

Таким образом, задача проверки эквивалентности схем сводится к решению SAT-задачи:

$$\exists x: M(x) = 1\ ?$$

На практике булева функция $M(x)$ задаётся в виде логической формулы (например, в конъюнктивной нормальной форме), которая передаётся на вход SAT-решателю. Если решатель возвращает статус SAT, это означает, что найден вход, на котором схемы различаются, и эквивалентность нарушена. Если же формула оказывается UNSAT, то выводится, что схемы эквивалентны.

В настоящей работе в программной реализации рассматривается одновыходной случай ($m = 1$), то есть каждая схема задаётся одной булевой функцией $f(x)$. Это соответствует, например, проверке эквивалентности

отдельных выходных линий или внутреннего сигнала схемы. Обобщение на многовыходный случай $m > 1$ выполняется стандартно путём дизъюнкции XOR-различий по всем выходам, однако в данной версии программного модуля векторные выходы не поддерживаются и используются только скалярные формулы.

Сложность построения митора.

Построение митора для одного входа $M = (F_1 \wedge \neg F_2) \vee (\neg F_1 \wedge F_2)$ на уровне AST требует добавления константного числа узлов поверх деревьев F_1 и F_2. Поэтому размер митора удовлетворяет оценке $|M| = O(|F_1| + |F_2|)$, а время его построения и преобразования в формат решателя (например, AST→Z3) линейно по размеру $|M|$.

При использовании КНФ-преобразования (вспомогательные переменные для подформул) размер КНФ также растёт линейно: число новых переменных и клауз $O(|M|)$.

Несмотря на линейность построения, итоговая производительность определяется в основном трудоёмкостью SAT-решения, которая зависит от числа переменных/клауз и структуры полученной формулы. Таким образом, построение митора обычно не является “узким местом”, однако оно влияет на размер и связность задачи, что отражается на времени работы SAT-решателя.

1.4. Алгоритмы SAT-решения: DPLL, CDCL. Особенности Z3

Использование SAT-подхода для верификации комбинационных схем опирается на существование эффективных алгоритмов решения задачи выполнимости булевых формул. В отличие от полного перебора по всем входным векторам, SAT-решатели используют структуру формулы и эвристики поиска, что позволяет на практике обрабатывать формулы с тысячами и миллионами переменных. В данном разделе кратко рассматриваются классический алгоритм DPLL, его развитие в виде CDCL-решателей, а также особенности решателя Z3, используемого в работе.

В качестве входа SAT-решатель получает булеву формулу, как правило, приведённую к конъюнктивной нормальной форме (КНФ). Задача состоит в определении, существует ли присваивание переменным, при котором формула принимает значение «истина». При проверке эквивалентности схем в данной работе SAT-решателю передаётся формула митора, выражающая несоответствие выходов двух схем (подробно в разд. 1.3), поэтому ответ SAT трактуется как «существует контрпример эквивалентности».

1.4.1. Классический алгоритм DPLL

Исторически одним из первых эффективных алгоритмов решения задачи выполнимости булевых формул в конъюнктивной нормальной форме (КНФ) стал алгоритм Дэвиса-Патнэма-Логемана-Лавленда (DPLL). Он работает с формулой вида

$$F = C_1 \wedge C_2 \wedge \ldots \wedge C_m,$$

где каждая клауза C_j – дизъюнкция литералов вида x_i или $\overline{x_i}$.

Идея DPLL заключается в рекурсивном поиске с возвратом (*backtracking*) по пространству частичных присваиваний переменным с использованием нескольких правил упрощения.

Основные шаги алгоритма:

1. *Unit propagation* (распространение единичных клауз).

Если в формуле есть клауза, состоящая из одного литерала ($\boldsymbol{\ell}$), то этот литерал обязан быть истинным в любой модели. Поэтому:

- переменной, соответствующей $\boldsymbol{\ell}$, присваивается значение, делающие $\boldsymbol{\ell}$ истинным;
- все клаузы, содержащие $\boldsymbol{\ell}$, считаются удовлетворёнными и удаляются;
- из всех клауз, содержащих $\overline{\boldsymbol{\ell}}$, этот литерал удаляется.

Процесс повторяется до тех пор, пока новые единичные клаузы не появляются или не возникает пустая клауза (противоречие).

2. Правило чистого литерала.

Если некоторая переменная встречается в формуле только с одним знаком (т.е. в виде x_i, но нигде не встречается $\overline{x_i}$, или наоборот), то её можно зафиксировать соответствующим образом так, чтобы все клаузы с этим литералом стали истинными. Удаление таких клауз упрощает формулу без изменения её выполнимости.

3. Ветвление по переменной (*decision*).

Если дальнейшие упрощения невозможны, выбирается ещё не назначенная переменная x_k и делается гипотетическое решение, например $x_k = 1$. После этого к формуле рекурсивно применяется DPLL с учётом нового частичного присваивания.

4. Возврат (*backtracking*).

Если в процессе рекурсивного решения возникает конфликт (формула становится невыполнимой при текущем наборе решений), алгоритм возвращается назад и пробует противоположное значение $x_k = 0$. Если и оно приводит к конфликту, выполняется возврат ещё на один уровень и меняется предыдущее решение.

Алгоритм DPLL гарантированно завершает работу и эквивалентен по выразительной мощности полному перебору, но благодаря *unit propagation* и правилу чистого литерала он отсекает большую часть пространства поиска ещё до явного перебора всех комбинаций. Однако в худшем случае его временная сложность остаётся экспоненциальной.

В рамках работы реализована учебная версия DPLL для решения CNF и сопоставления с другими подходами. Для практической производительности дополнительно используется промышленный решатель Z3.

1.4.2. Современные CDCL-решатели

Современные промышленные SAT-решатели реализуют развитие идеи DPLL, известное как CDCL (Conflict-Driven Clause Learning – поиск с обучением по конфликтам). CDCL-алгоритмы сохраняют базовую структуру

DPLL (ветвления + *unit propagation* + возврат), но добавляют ряд ключевых механизмов, существенно повышающих эффективность:

- Обучение по конфликтам (*clause learning*).

При возникновении конфликта решатель анализирует последовательность решений и пропагаций, приведших к противоречию, и на основе этого строит новую клауза – так называемую конфликтную клауза. Эта клауза добавляется к формуле и запрещает повторно проходить тот же конфликтный путь. Таким образом, решатель накапливает информацию о «плохих» частях пространства поиска.

- Нехронологический возврат (*non-chronological backtracking*).

В отличие от классического DPLL, где возврат происходит строго к последнему решению, CDCL может возвращаться сразу к более раннему уровню, непосредственно связанному с причиной конфликта. Это позволяет перескакивать через целые уровни решений, которые не влияли на возникновение противоречия.

- Эвристики выбора переменных.

CDCL-решатели используют эвристики активности (например, семейство VSIDS), которые предпочитают делать решения по тем переменным, что чаще всего фигурируют в конфликтах. Это помогает быстрее находить информативные конфликтные клаузы и ускоряет сужение пространства поиска.

VSIDS / EVSIDS (эвристики активности)

Наиболее распространённый класс эвристик — VSIDS (*Variable State Independent Decaying Sum*) и его модификации (EVSIDS). Каждой переменной xxx сопоставляется числовой параметр активности $A(x)$. При возникновении конфликта и построении выученной клаузы активность переменных, вошедших в эту клаузы, увеличивается (операция *bump*). Далее активности подвергаются затуханию (*decay*): вклад давних конфликтов постепенно уменьшается, а свежие конфликты оказываются важнее. Переменная для

следующего решения выбирается как переменная с максимальной активностью среди не назначенных. Интуитивно это означает, что решатель концентрируется на переменных, которые чаще всего оказываются вовлечены в противоречия и, следовательно, лучше всего “разрезают” пространство поиска.

- Перезапуски (*restarts*).

В процессе решения решатель периодически перезапускает поиск, сохраняя при этом выученные клаузы. Это позволяет выходить из неудачных участков пространства решений, не теряя накопленной информации о конфликтах.

- Предобработка и *in-processing*.

Дополнительно применяются различные техники упрощения формулы: удаление поглощённых клауз, устранение эквивалентных и чистых литералов, частичное устранение переменных и др. Часть таких преобразований выполняется до запуска основного поиска, часть – динамически в ходе работы решателя.

Благодаря этим усовершенствованиям CDCL-решатели способны эффективно обрабатывать многие практические SAT-задачи, которые были бы совершенно недоступны для прямого DPLL или тем более для полного перебора. Именно такого класса алгоритмы лежат в основе используемого в работе решателя Z3.

Сложность алгоритма CDCL.

CDCL является полным методом решения SAT и, как развитие DPLL, в худшем случае сохраняет экспоненциальную сложность по числу переменных nnn: в общем случае возможен перебор $O(2^n)$, что обусловлено NP-полнотой задачи SAT. Следовательно, универсальной полиномиальной оценки времени работы для CDCL не существует.

Практическая эффективность CDCL объясняется использованием обучения по конфликтам (clause learning), нехронологического возврата

(backjumping), эвристик выбора переменных и стратегии перезапусков. Эти механизмы уменьшают повторение одних и тех же конфликтных ситуаций и часто приводят к резкому сокращению объёма поиска на структурированных задачах, характерных для верификации схем.

В рамках данной работы эффективность CDCL оценивается экспериментально путём измерения времени решения на серии тестовых формул/миторных задач и сравнения с DPLL, Z3 и полным перебором (на малых размерах).

1.4.3. Особенности решателя Z3 и его использование в работе

Z3 представляет собой промышленный SMT-решатель общего назначения, разработанный компанией Microsoft. В частности, он включает эффективное SAT-ядро, основанное на CDCL-алгоритмах. В настоящей работе используется только булев фрагмент Z3, то есть решатель применяется как высокопроизводительный SAT-решатель.

1. Использование Z3 как «чёрного ящика»

Программный модуль, реализованный в работе, поддерживает несколько вариантов SAT-проверки эквивалентности. В качестве базового решения используется промышленный решатель Z3: формула митора строится на уровне абстрактного синтаксического дерева (AST) и затем отображается в булевы выражения Z3. На языке Python это выглядит следующим образом: сначала исходные формулы `f1_expr` и `f2_expr` разбираются в AST, затем с помощью функции `ast_to_z3()` отображаются в булевы выражения Z3, после чего конструируется митор

$$D(x) = \left(F_1(x) \wedge \neg F_2(x)\right) \vee \left(\neg F_1(x) \wedge F_2(x)\right)$$

соответствующий коду

```
miter = or(and(f1, not(f2)), and(not(f1), f2))
```

Эта формула добавляется в объект `Solver`, и далее вызывается метод `check()`. Результат `sat` интерпретируется как существование входа, на

котором схемы различаются, результат `unsat` – как доказательство их эквивалентности.

2. Встроенные алгоритмы предобработки.

Z3 самостоятельно выполняет необходимые преобразования формулы: преобразование к внутреннему представлению, частичное преобразование к КНФ, *unit propagation*, устранение очевидных противоречий и другие оптимизации. В рамках данной работы эти шаги не реализуются явно в пользовательском коде, а полностью делегируются решателю.

3. Сравнение с полным перебором.

- традиционный метод полного перебора, реализованный функцией `check_equiv_bruteforce()`, которая перебирает все 2^n комбинаций входов до ограничения по числу переменных;
- SAT-подход с использованием Z3, реализованный функцией `check_equiv_z3()`.

Бенчмарк-скрипт benchmark.py генерирует случайные пары булевых формул, измеряет время работы обоих методов и строит зависимость среднего времени от числа переменных.

Помимо использования Z3, в рамках работы реализованы упрощённые (учебные) версии алгоритмов DPLL и CDCL. Для их применения митор переводится в конъюнктивную нормальную форму (КНФ) с введением вспомогательных переменных (Tseitin-подобное преобразование), что позволяет избежать экспоненциального разрастания формулы. После этого выполняется SAT-поиск: DPLL использует единичное распространение и возврат с перебором решений, а CDCL дополняет этот процесс анализом конфликтов и обучением клауз. Реализации DPLL/CDCL используются главным образом для иллюстрации принципов SAT-решения и сравнительного анализа с промышленным решателем Z3.

1.5. Выводы по первой главе

В первой главе были рассмотрены теоретические основы верификации комбинационных логических схем с использованием SAT-подхода. Показано, что поведение комбинационной схемы однозначно задаётся соответствующей булевой функцией, определяющей отображение из множества входных векторов $\{0,1\}^n$ в множество выходных векторов $\{0,1\}^m$. Это позволяет переходить от структурного описания схем на уровне логических элементов к более абстрактному представлению в виде логических формул над входными переменными.

Сформулирована задача проверки эквивалентности двух комбинационных схем как задача установления тождества двух булевых функций: схемы считаются эквивалентными, если для любого входного вектора их выходные значения совпадают. Было отмечено, что прямой подход, основанный на полном переборе всех возможных входных наборов, обладает экспоненциальной сложностью по числу входных переменных и быстро становится вычислительно неэффективным при росте размера схем.

Показано, что задача проверки эквивалентности может быть сведена к задаче выполнимости булевой формулы (SAT) с использованием конструкции митора. Для пары схем строится формула, описывающая ситуацию различия их выходов; выполнимость этой формулы означает существование контрпримера эквивалентности, а невыполнимость – эквивалентность схем. Таким образом, задача верификации схем переходит в задачу SAT, для которой существуют специализированные алгоритмы и программные решатели.

Рассмотрены основные алгоритмические подходы к решению задач SAT. Описан классический алгоритм DPLL, сочетающий поиск с возвратом и логические упрощения (*unit propagation*, правило чистого литерала), а также его развитие в виде современных CDCL-решателей, использующих обучение по конфликтам, не хронологический возврат, развитые эвристики выбора переменных и перезапуски. Отмечено, что данные расширения позволяют

существенно повысить практическую эффективность SAT-решателей по сравнению с наивным перебором.

Отдельное внимание уделено промышленным SAT-решателям на примере Z3, который в данной работе используется как базовый инструмент SAT-проверки и ориентир для сравнения. Z3 рассматривается как «чёрный ящик», реализующий современные алгоритмы класса CDCL и внутренние оптимизации; на стороне пользователя формируется булева формула митора, представляющая задачу проверки эквивалентности. Вместе с тем, для более наглядного понимания принципов SAT-решения в рамках работы также рассматриваются алгоритмы DPLL и CDCL как алгоритмическая основа, что позволяет сопоставлять теоретические идеи SAT-поиска с практической проверкой эквивалентности схем. Для учебных реализаций DPLL/CDCL задача дополнительно приводится к КНФ, тогда как при использовании Z3 формула задаётся напрямую на уровне булевых выражений.

2. РАЗРАБОТКА ПРОГРАММНОГО МОДУЛЯ ПРОВЕРКИ ЭКВИВАЛЕНТНОСТИ СХЕМ

2.1. Постановка требований к программному модулю и выбор инструментальных средств

Целью разработки является программный модуль, реализующий четыре подхода к проверке эквивалентности комбинационных схем:

- традиционный метод полного перебора входных векторов;
- метод на основе SAT-решателя Z3 с использованием митора;
- учебный SAT-решатель на базе алгоритма DPLL (работа с КНФ митора);
- учебный SAT-решатель на базе алгоритма CDCL (работа с КНФ митора и обучением клауз).

На основе теоретической части формулируются следующие основные требования к программному обеспечению:

1. Поддержка удобного представления схем.

Схема должна задаваться пользователем в виде булевой формулы над переменными $x_0, x_1, ...$ с использованием стандартных логических операций: отрицание (!), конъюнкция (&), дизъюнкция (|), импликация (->), эквивалентность (<->) и круглые скобки. Это позволяет описывать схемы на уровне выражений, а не явных вентильных сетей.

2. Проверка эквивалентности четырьмя независимыми методами.

Для двух заданных формул F_1, F_2 модуль должен уметь:

- проверить их эквивалентность полным перебором всех входных наборов (при разумном ограничении числа переменных);
- построить формулу митора, передать её в SAT-решатель Z3 и по результату SAT/UNSAT сделать вывод об эквивалентности;
- построить митор и перевести его в КНФ для запуска учебного решателя DPLL;

- построить митор и перевести его в КНФ для запуска учебного решателя CDCL.

3. Измерение времени работы методов.

Для экспериментального сравнения необходимо измерять и возвращать время работы каждого метода. В программе функции проверки эквивалентности возвращают пару «результат проверки, время работы в миллисекундах». Для SAT-методов измеряемая величина интерпретируется как время работы соответствующего этапа SAT-проверки (решателя), что позволяет сопоставлять производительность подходов.

4. Ограничения по масштабу задач.

Поскольку сложность полного перебора экспоненциальна по числу входных переменных, в программе задаётся явный предел на размер задачи для брутфорса. В текущей реализации при числе переменных более 25 функция `check_equiv_bruteforce()` возбуждает исключение, и полный перебор для такой пары формул не выполняется. Это позволяет избежать заведомо непрактичных вычислений.

5. Возможность автоматического бенчмаркинга.

Программный комплекс должен поддерживать автоматическую генерацию случайных булевых формул, многократный запуск обоих методов на этих данных и сохранение результатов в виде таблицы (CSV) для последующей визуализации.

Исходя из перечисленных требований, были выбраны следующие инструментальные средства:

- Язык программирования Python.

Python обеспечивает высокую скорость разработки, удобную работу со строками, удобные средства построения и обработки абстрактных синтаксических деревьев, а также богатую экосистему библиотек для научных вычислений и визуализации. Хотя Python уступает компилируемым языкам по скорости, в данной работе критично не абсолютное время исполнения, а

сравнение относительной эффективности методов (брутфорса и Z3) на одних и тех же задачах.

- Библиотека z3-solver для доступа к решателю Z3.

Z3 представляет собой промышленный SMT/SAT-решатель, для которого существует официальный Python-интерфейс. В программе он используется как SAT-решатель: на базе типов Bool, операций And, Or, Not формируется формула митора, которая затем передаётся в объект `Solver` для проверки выполнимости. Такой подход позволяет использовать внутри программы всю мощь современных CDCL-алгоритмов без необходимости реализовывать их самостоятельно.

- Библиотека matplotlib для построения графиков.

Данная библиотека применяется в модуле plot_benchmark.py для построения графика зависимости среднего времени работы методов от числа входных переменных. Визуализация результатов является важной частью экспериментального раздела.

Таким образом, выбранные средства позволяют удовлетворить функциональные требования к модулю проверки эквивалентности схем, при этом упрощая реализацию и сосредотачивая усилия на логической постановке задачи и корректности преобразований, а не на низкоуровневой оптимизации алгоритмов SAT-решения.

2.2. Формат представления входных данных и модуль разбора булевых формул

Как отмечалось в теоретической части, в рамках данной работы комбинационная схема представляется в виде булевой формулы над входными переменными. В программной реализации пользователь задаёт две схемы в текстовом виде, а сама программа выполняет разбор этих строк, построение внутреннего представления и дальнейшие вычисления.

2.2.1. Синтаксис булевых формул

Входная формула задаётся как строка, содержащая:

- имена переменных вида x0, x1, x2, …;
- логические операции:
  - ! – отрицание (унарный оператор);
  - & – конъюнкция;
  - | – дизъюнкция;
  - -> – импликация;
  - <-> – эквиваленция;
- круглые скобки (и) для явного задания приоритета операций;
- произвольные пробелы, которые игнорируются при разборе.

Приоритет операций задаётся следующим образом: ! > & > | > → > ↔

Это означает, что отрицание имеет наивысший приоритет, затем идут конъюнкция и дизъюнкция, а импликация и эквивалентность связывают подвыражения на самом верхнем уровне. Такой порядок обеспечивает интуитивно ожидаемое поведение при отсутствии скобок.

2.2.2. Лексический анализ (функция tokenize)

Первым этапом обработки входной строки является лексический анализ – разбиение исходного текста на последовательность токенов (лексем). В программе эта задача выполняется функцией `tokenize(expr: str) -> List[str]`, которая последовательно просматривает символы строки и выделяет:

- идентификаторы переменных (x0, x1, a, b и т.п.), представленные как последовательности букв, цифр и подчёркивания;
- односимвольные операторы и скобки: (,), !, &, |;
- двух- и трёхсимвольные операторы: ->, <->

Функция игнорирует пробельные символы и выбрасывает исключение SyntaxError при встрече неожиданных символов. В результате строка вида

`!(x0 & x1) | x2` преобразуется в список токенов: `["!", "(", "x0", "&", "x1", ")", "|", "x2"]`. Такое представление удобно для последующего синтаксического анализа.

2.2.3. Структура абстрактного синтаксического дерева

Для дальнейшей обработки формула представляется в виде абстрактного синтаксического дерева (AST). В программе используется три типа узлов, реализованных через `@dataclass`:

- `Var` – переменная:

```
@dataclass
class Var:
    name: str
```

- `NotNode` – отрицание:

```
@dataclass
class NotNode:
    child: "Node"
```

- `BinNode` – бинарная операция:

```
@dataclass
class BinNode:
    op: str
    left: "Node"
    right: "Node"
```

Общий тип узла задаётся как объединение:

```
Node = Union[Var, NotNode, BinNode]
```

Такое дерево является удобной универсальной формой представления формулы: на его основе можно как вычислять значение выражения для заданного набора переменных (что делается косвенно через `ast_to_python()` и `make_circuit_lambda()`), так и строить эквивалентное выражение для Z3 (`ast_to_z3()`).

2.2.4. Синтаксический анализ

Синтаксический анализ реализован функцией `parse(expr: str) -> Node`. Она вызывает лексер `tokenize`, а затем использует рекурсивный нисходящий парсер, который учитывает приоритеты операций.

Внутри `parse` создаётся список токенов и переменная `pos`, указывающая на текущую позицию. Затем определяются вложенные функции:

- `peek()` – возвращает текущий токен или `None` при конце списка;
- `take(expected)` – потребляет ожидаемый токен или выбрасывает SyntaxError;
- `parse_expr()` – стартовая функция, вызывающая `parse_equiv()`.

Парсер построен по иерархии приоритетов:

- `parse_equiv()` – обрабатывает цепочки с оператором <->;
- `parse_impl()` – операцию ->;
- `parse_or()` – операции |;
- `parse_and()` – операции &;
- `parse_unary()` – унарное отрицание !;
- `parse_atom()` – скобочные выражения и переменные.

Например, при разборе выражения `x0 & (x1 | !x2)` парсер последовательно:

- распознаёт верхнеуровневую конъюнкцию `x0 & (…)` в `parse_and()`;
- в качестве левого операнда строит узел `Var("x0")`;
- в качестве правого – вызывает `parse_unary() -> parse_atom()`, где обрабатывается конструкция в скобках `(x1 | !x2)` как бинарный узел `BinNode("|", Var("x1"), NotNode(Var("x2")))`.

В случае синтаксической ошибки (незаписанная закрывающая скобка, неожиданный токен и т.п.) выбрасывается исключение SyntaxError, которое используется, в частности, в модуле manual_test.py для вывода понятного сообщения пользователю.

Таким образом, модуль разбора формул обеспечивает переход от текстового представления схем к структурированному внутреннему виду (AST), на основе которого в последующих подпунктах реализуются оба метода проверки эквивалентности: традиционный (полный перебор) и SAT-подход с использованием Z3.

2.3. Реализация традиционного метода проверки эквивалентности (полный перебор)

Традиционный подход к проверке эквивалентности двух комбинационных схем заключается в полном переборе всех возможных входных векторов и сравнении значений выходных функций. В программной реализации данный метод инкапсулирован в функции `check_equiv_bruteforce(f1_expr: str, f2_expr: str) -> Tuple[bool, float]` модуля main.py.

На вход функция получает две строки `f1_expr` и `f2_expr`, задающие булевы формулы, соответствующие сравниваемым схемам. На первом шаге из текстового представления извлекается множество входных переменных. Для этого используется вспомогательная функция `extract_vars(expr: str) -> List[str]`, которая просматривает строку и собирает все идентификаторы вида `x0, x1, …, xN`. Переменные сортируются по числовому индексу, что обеспечивает фиксированный порядок при переборе.

Далее каждая формула преобразуется в исполняемую функцию от словаря значений переменных. Для этого применяется композиция функций `parse()`, `ast_to_python()` и `make_circuit_lambda()`. Сначала строковое выражение разбирается в абстрактное синтаксическое дерево (`Node`), затем дерево преобразуется в эквивалентное Python-выражение логики через `ast_to_python()`, и наконец это выражение компилируется и оборачивается в функцию:

```
def make_circuit_lambda(expr: str) ->
Callable[[Dict[str, bool]], bool]:
    ast = parse(expr)
    expr_py = ast_to_python(ast)
    code = compile(expr_py, "", "eval")

    def circuit(assign: Dict[str, bool]) -> bool:
        return bool(eval(code, {"__builtins__": {}},
assign))

    return circuit
```

Таким образом, для заданных формул `f1_expr` и `f2_expr` строятся две функции `f1(env)` и `f2(env)`, которые по словарю `env` вида `{"x0": True, "x1": False, ...}` возвращают значение соответствующей булевой функции.

После подготовки функций начинается собственно полный перебор. Если формулы не содержат переменных (константные выражения), то достаточно одного вызова `f1({})` и `f2({})`. В общем случае, при наличии n переменных, количество возможных входных наборов равно 2^n. В коде перебор реализован с помощью функции `itertools.product()`:

```
for bits in product([False, True], repeat=n):
    env = {name: bit for name, bit in zip(all_vars,
bits)}
    if f1(env) != f2(env)
        # найден контрпример
        ...
```

Для каждого вектора `bits` формируется словарь `env`, после чего вычисляются значения обеих функций. Если хотя бы на одном наборе входов результаты различаются, формулы (а значит и схемы) признаются неэквивалентными, и поиск немедленно прерывается. Время работы измеряется с помощью `time.perf_counter()`, и функция возвращает пару: логическое значение «эквивалентны/неэквивалентны» и затраченное время в миллисекундах.

Поскольку сложность полного перебора экспоненциально растёт с числом входных переменных, в реализации предусмотрено ограничение на размер задачи: при $n > 25$ функция возбуждает исключение `RuntimeError("Too many inputs for brute force")`. Это соответствует теоретическому выводу о практической неприменимости полного перебора для схем с большим числом входов и одновременно защищает программу от чрезмерно длительных вычислений.

2.4. Реализация SAT-подхода

В рамках SAT-подхода проверка эквивалентности двух схем (формул) F_1 и F_2 сводится к проверке выполнимости формулы митора:

$$M = F_1 \oplus F_2 = (F_1 \wedge \neg F_2) \vee (\neg F_1 \wedge F_2).$$

Формула MMM истинна ровно на тех наборах входных переменных, где F_1 и F_2 дают разные значения. Поэтому:

- если MMM выполнима (SAT), существует контрпример, и схемы неэквивалентны;
- если MMM невыполнима (UNSAT), контрпримеров нет, и схемы эквивалентны.

Ниже описаны три реализации SAT-проверки: через Z3 (промышленный решатель) и две учебные реализации SAT-решателей (DPLL и CDCL), работающие с КНФ.

2.4.1. Проверка эквивалентности с использованием решателя Z3

Ветвь Z3 реализована функцией `check_equiv_z3(f1_expr: str, f2_expr: str) -> Tuple[bool, float]`.

Как и в случае полного перебора, на вход подаются две строки с булевыми формулами. Сначала из них извлекается совокупность переменных через `extract_vars()`, после чего для каждой переменной создаётся соответствующая булева переменная Z3:

```
vars1 = extract_vars(f1_expr)
vars2 = extract_vars(f2_expr)
all_vars = sorted(set(vars1) | set(vars2), key=lambda
s: int(s[1:]))

ctx_vars: Dict[str, Bool] = {name: Bool(name) for name
in all_vars}
```

Далее обе формулы последовательно разбираются и отображаются в выражения языка Z3. Эту задачу выполняют функции `parse()` и `ast_to_z3()` (обёрнутая в `build_z3_circuit()`):

```
def build_z3_circuit(expr: str, ctx_vars: Dict[str,
Bool]):
    ast = parse(expr)
    return ast_to_z3(ast, ctx_vars)
```

Функция `ast_to_z3()` рекурсивно обходится по дереву `Node` и заменяет его узлы на соответствующие операции Z3: And, Or, Not, а также эквивалентные представления импликации и эквивалентности:

```
if node.op == "&":
    return And(l, r)
if node.op == "|":
    return Or(l, r)
if node.op == "->":
    return Or(Not(l), r)
if node.op == "<->":
    return Or(And(l, r), And(Not(l), Not(r)))
```

После построения двух булевых выражений `f1` и `f2` формируется митор:

```
miter = Or(And(f1, Not(f2)), And(Not(f1), f2))
```

Эта формула истинна в точности на тех наборах входных переменных, где значения F1 и F2 различаются. Далее создаётся объект решателя Z3:

```
s = Solver()
s.add(miter)
```

и запускается проверка выполнимости:

```
start = time.perf_counter()
res = s.check()
dt = (time.perf_counter() - start) * 1000.0
```

Если решатель возвращает `sat`, это означает, что существует входной вектор, при котором `miter` истинна, то есть схемы неэквивалентны. В противном случае (`unsat`) делается вывод об их эквивалентности. В реализации логическое значение результата сформулировано как:

```
return (res != sat), dt
```

Таким образом, функция `check_equiv_z3()` также возвращает два значения: факт эквивалентности и время работы, но уже без явного перебора входных наборов. Вся сложность SAT-поиска, включая DPLL/CDCL-алгоритмы, предобработку и эвристики, полностью скрыта внутри решателя Z3 и не реализуется в пользовательском коде.

Важно отметить, что обе процедуры – полный перебор и SAT-подход – используют общий модуль разбора формул и единое внутреннее представление (AST). Это гарантирует, что различия в результатах и времени работы обусловлены именно алгоритмом проверки эквивалентности, а не различиями в интерпретации входных данных.

2.4.2. Учебная реализация DPLL (митор → КНФ → DPLL)

Помимо использования промышленного решателя Z3, в работе реализован учебный SAT-решатель на основе классического алгоритма DPLL. В отличие от Z3, DPLL работает с формулой в конъюнктивной нормальной форме (КНФ), то есть в виде конъюнкции клауз:

$$C_1 \wedge C_2 \wedge \ldots \wedge C_k,$$

где каждая клауза C_i представляет собой дизъюнкцию литералов.

В программном модуле данный вариант реализован функцией

```
check_equiv_dpll(f1_expr: str, f2_expr: str) -> Tuple[bool, float]
```

Построение КНФ митора

На первом этапе формулы F_1 и F_2 разбираются в AST. Затем строится AST митора:

$$M = (F_1 \wedge \neg F_2) \vee (\neg F_1 \wedge F_2).$$

Далее выполняется перевод митора в КНФ функцией `ast_to_cnf()`. В ходе преобразования используется введение вспомогательных переменных вида `_aux1, _aux2,` …, которые соответствуют значениям подформул. Такой подход позволяет избежать экспоненциального роста формулы при прямом раскрытии скобок и обеспечивает линейный (по размеру AST) рост числа клауз.

Полученная КНФ представляется структурой `Formula`, содержащей список объектов `Clause`, а каждая клауза содержит список `Literal`. Дополнительно добавляется единичная клауза, требующая истинности корневой переменной, соответствующей миторной формуле, что эквивалентно условию $M = \text{true}$.

Алгоритм DPLL

После построения КНФ запускается решатель `dpll_solve_cnf(formula)`. Алгоритм DPLL реализует поиск с возвратом и включает следующие ключевые этапы:

1. Unit propagation (единичное распространение).

Если в формуле присутствует единичная клауза вида (l), то литерал l обязан быть истинным, иначе формула станет невыполнимой. Поэтому переменная из литерала фиксируется принудительно, после чего формула упрощается.

2. Выбор переменной и ветвление.

Если не все переменные назначены и конфликт не обнаружен, выбирается некоторая ещё не назначенная переменная и ей задаётся значение (в учебной реализации – простая стратегия выбора).

3. Возврат (backtracking).

Если в процессе упрощения возникает конфликт (какая-либо клауза становится ложной), выполняется откат к предыдущему решению и пробуется альтернативное значение выбранной переменной.

Интерпретация результата

- если DPLL находит удовлетворяющее присваивание (SAT), то митор выполним, существует контрпример, и схемы неэквивалентны;
- если алгоритм исчерпывает поиск и возвращает отсутствие модели (UNSAT), то митор невыполним, и схемы эквивалентны.

Время выполнения измеряется вокруг вызова `dpll_solve_cnf()`, что позволяет оценивать трудоёмкость работы учебного решателя на КНФ митора.

2.4.3. Учебная реализация CDCL (митор → КНФ → CDCL)

Для более приближенного к промышленным SAT-решателям подхода реализован учебный вариант алгоритма CDCL (Conflict-Driven Clause Learning). В сравнении с DPLL, CDCL использует анализ конфликтов, добавление “выученных” клауз и нехронологический возврат, что позволяет существенно сокращать объём перебора.

Данный вариант реализован функцией `check_equiv_cdcl(f1_expr: str, f2_expr: str) -> Tuple[bool, float].`

Затем строится КНФ митора, аналогично тому, как это делалось для DPLL.

Основные элементы CDCL

Далее запускается `cdcl_solve_cnf(formula)`. В учебной реализации выделяются следующие ключевые механизмы:

1. Unit propagation с сохранением причин.

Для каждого принудительного присваивания запоминается клауза-причина (antecedent), из-за которой переменная была обязана принять конкретное значение. Это необходимо для последующего анализа конфликтов.

2. Обнаружение конфликта.

Если во время propagation появляется клауза, все литералы которой ложны, фиксируется конфликтная клауза.

3. Анализ конфликта (conflict analysis) и обучение (learning).

По цепочке причин строится новая «выученная» клауза (learnt clause), которая запрещает повторение того же конфликта при дальнейшем поиске.

4. Нехронологический возврат (backjump).

После обучения выполняется откат не обязательно на последний уровень решения, а на более ранний уровень, вычисленный из структуры конфликта. Это отличается от обычного backtracking в DPLL и ускоряет поиск.

После добавления выученной клаузы поиск продолжается до нахождения модели или доказательства невыполнимости.

Интерпретация результата

Интерпретация совпадает с общей схемой SAT-подхода:

- SAT → найден контрпример → схемы неэквивалентны;
- UNSAT → контрпримеров нет → схемы эквивалентны.

Измерение времени производится вокруг вызова `cdcl_solve_cnf()`, что позволяет сравнивать эффективность CDCL и DPLL на одном и том же классе задач (КНФ митора).

2.5. Структура программного комплекса и вспомогательные модули

Разработанный программный комплекс состоит из четырёх основных файлов: main.py, manual_test.py, benchmark.py и plot_benchmark.py.

В файле main.py сосредоточена основная логика. В нём определены структуры данных для представления булевых формул в виде абстрактного синтаксического дерева (AST), реализованы лексический анализ и синтаксический разбор входных выражений, а также процедуры преобразования формул в различные представления. На базе этого реализованы ключевые методы проверки эквивалентности: `check_equiv_bruteforce()` (полный перебор всех входных наборов при заданном ограничении), `check_equiv_z3()` (построение митора и проверка выполнимости с помощью Z3), а также учебные решатели `check_equiv_dpll()` и `check_equiv_cdcl()`, которые решают SAT-задачу

для митора после приведения её к КНФ. В этом же модуле реализованы функции генерации случайных формул и демонстрационная функция `demo`, позволяющая запускать серию тестов и сравнивать результаты методов.

Модуль manual_test.py обеспечивает интерактивный режим работы. Пользователь вводит две формулы в командной строке, после чего программа вызывает функции из main.py, выполняет проверку эквивалентности выбранными методами и выводит на экран вердикт и измеренное время работы. Такой режим удобен для проверки отдельных примеров и демонстрации работы программы.

Модуль benchmark.py предназначен для автоматического бенчмаркинга. Он генерирует тестовые пары булевых формул при различных параметрах (например, число переменных и глубина выражений), запускает проверку эквивалентности и сохраняет результаты измерений в табличном виде (CSV) для последующего анализа. В зависимости от выбранной конфигурации бенчмаркинг может включать как сравнение базовых подходов (полный перебор и Z3), так и дополнительные SAT-реализации.

Наконец, модуль plot_benchmark.py читает файл с результатами бенчмарка и строит графики зависимости среднего времени работы методов от параметров задачи (например, числа входных переменных). Это позволяет наглядно сравнить масштабируемость и практическую эффективность различных подходов.

В совокупности указанные модули образуют целостный программный комплекс: от реализации методов проверки эквивалентности до удобных средств тестирования и экспериментального анализа.

2.6. Выводы по второй главе

Во второй главе была описана реализация программного модуля для проверки эквивалентности комбинационных схем.

Сначала были сформулированы требования к системе и обоснован выбор инструментов: язык Python и библиотека z3-solver для работы с

SAT-решателем Z3. Далее определён формат задания схем в виде логических формул над переменными x0, x1, … и реализован модуль разбора: лексический анализ входной строки, построение абстрактного синтаксического дерева и его дальнейшая обработка.

На этой основе реализованы методы проверки эквивалентности, основанные на различных подходах. Во-первых, реализован традиционный метод полного перебора, который вычисляет значения обеих формул на всех возможных входных наборах (до заданного ограничения по числу переменных). Во-вторых, реализован SAT-подход на базе решателя Z3: для двух формул строится митор, переводится во внутреннее представление Z3 и проверяется на выполнимость, что позволяет избегать явного перебора входных наборов. Дополнительно реализованы учебные SAT-решатели на основе алгоритмов DPLL и CDCL, применяемые к КНФ-представлению митора, что позволяет сопоставить классические идеи SAT-поиска с практической проверкой эквивалентности.

Также была описана структура программного комплекса: основной модуль с реализацией логики и методов, интерактивный модуль для ручного тестирования и модули для автоматического бенчмаркинга и визуализации результатов. Это создаёт основу для последующего экспериментального сравнения подходов в следующей главе.

Таким образом, во второй главе разработан и описан программный инструментарий, обеспечивающий проверку эквивалентности схем несколькими методами и поддерживающий проведение вычислительных экспериментов и анализ результатов.

3. ЭКСПЕРИМЕНТАЛЬНОЕ ИССЛЕДОВАНИЕ ЭФФЕКТИВНОСТИ МЕТОДОВ

3.1. Постановка эксперимента

В третьей главе на основе разработанного программного комплекса проводится количественное и качественное сравнение описанных ранее подходов к проверке эквивалентности комбинационных схем: традиционного метода полного перебора входных векторов, DPLL, CDCL и Z3.

Целью эксперимента является не только зафиксировать выигрыш SAT-метода по времени работы при увеличении числа входных переменных, но и убедиться в корректности программной реализации на наборе формул, для которых заранее известно, должны ли они быть эквивалентны или различаться, что особенно важно в контексте верификации схем, полученных различными путями синтеза и оптимизации.

3.2. Описание программных модулей

Экспериментальная часть опирается на два вспомогательных модуля: manual_test.py и benchmark.py, которые, используя общие функции `parse()`, `check_equiv_bruteforce()`, `check_equiv_z3()`, `check_equiv_dpll()` и `check_equiv_cdcl()` из основного модуля, обеспечивают соответственно интерактивные проверки и массовое автоматическое тестирование.

Скрипт manual_test.py реализует диалог с пользователем: функция `read_formula()` циклически запрашивает у него строковое выражение, сразу же прогоняет его через парсер и либо принимает формулу (при корректном синтаксисе), либо выводит диагностическое сообщение и повторяет запрос, что позволяет отлавливать ошибки скобок и операторов ещё до запуска решателей; далее функция `run_once()` выводит обе формулы, вызывает функции проверки четырьмя метододами, печатает для каждого метода логический результат (equiv/diff) и время в миллисекундах, а оболочка `main()`

организует многократный запуск этой процедуры до ввода пустой строки пользователем.

Модуль benchmark.py, напротив, полностью автоматизирует построение статистики: в функции `run_benchmark()` задаются диапазон числа переменных `NUM_VARS_LIST = list(range(2, 101, 2))` и число испытаний на каждую точку `TESTS_PER_POINT = 5`, после чего для каждого фиксированного n порождается серия из 5 пар формул функцией `generate_shuffled_chain()`, для каждой пары по очереди вызываются оба метода проверки, а измеренные времена аккумулируются в списках и усредняются, формируя строки вида (n, avg_brute, avg_dpll, avg_cdcl, avg_z3), которые в конце записываются в CSV-файл benchmark_results.

Количество переменных для полного перебора ограничено 23. Далее формулы проверяются оставшимися тремя методами, а брутфорс игнорируется.

Актуальная версия генератора формул представляет собой функцию `generate_shuffled_chain()`, которая в качестве аргумента принимает натуральное число n – длину формулы. Она создаёт последовательность x1, x2, …, xn, перемешивает её и соединяет методом join в одну строку с разделителем *&*. В итоге получаются тривиальные формулы конъюнкции, которые всегда эквивалентны. Благодаря этому подходу, мы получаем сравнение методов, которое зависит не от сложности формул, а только от их длины.

3.3. Ручные эксперименты

Использование интерактивного модуля позволило подобрать ряд показательных пар формул, отражающих как базовые законы булевой алгебры, так и более сложные соотношения между схемами. Сводные результаты по ним удобно представить в табл. 3.1, где для каждой пары приводятся сами формулы и измеренные времена работы методов. Также эти результаты можно посмотреть в файле manual_test_results.txt. Так, для

коммутативных и дистрибутивных тождеств (`x0 & x1` и `x1 & x0`, `(x0 & x1) | (x0 & x2)` и `x0 & (x1 | x2)`) оба метода неизменно возвращали «equiv», что подтверждает корректность.

Аналогично пары, отличающиеся одной операцией (`x0 & x1` и `x0 | x1`, `x0 & (x1 | x2)` и `(x0 & x1) | x2`, а также формулы, где дизъюнкция `x3 | x4` заменена на конъюнкцию `x3 & x4` или импликация `x4 -> x5` заменена на `x4 & x5`), во всех случаях были признаны неэквивалентными, что соответствует существованию конкретных входных векторов, на которых выходы схем различаются, и демонстрирует способность как полного перебора, так и SAT-подхода обнаруживать тонкие логические несоответствия в сложных формулах с пятью-шестью переменными.

Таблица 3.1

Результаты ручного тестирования

№	Формула F1	Формула F2	brute, мс	Z3, мс	DPLL, мс	CDCL, мс
1	`x0 & x1`	`x1 & x0`	0.012	60.02	0.209	0.370
2	`x0 & x1`	`x0 | x1`	0.008	7.522	0.125	0.169
3	`(x0 & x1) | (x0 & x2)`	`x0 & (x1 | x2)`	0.016	1.466	0.649	0.911
4	`(x0 & !x1) | (!x0 & x1)`	`!(x0 <-> x1)`	0.013	0.610	0.468	0.949
5	`x0 & (x1 | x2)`	`(x0 & x1) | x2`	0.010	0.419	0.240	0.392
6	`(x0 -> x1) & (x0 -> !x1)`	`!x0`	0.011	0.352	0.196	0.296
7	`(x0 <-> x1) & (x1 <-> x2)`	`(x0 <-> x2) & (x0 <-> x1)`	0.016	0.485	0.475	0.996
8	`x0 -> x1`	`!x0 | x1`	0.011	0.059	0.130	0.250
9	`(!x0 & x1) | (x0 & !x1) | (x2 & x3)`	`(!(x0 <-> x1)) | (x2 & x3)`	0.029	0.462	2.396	2.889
10	`((x0 | x1) & (!x2 | x3) & (x4 -> x5))`	`((x0 | x1) & (!x2 | x3) & (x4 & x5))`	0.026	0.498	0.885	1.223
11	`((x0 & x1) | (!x2 & (x3 | x4))) <-> x5`	`((x0 & x1) | (!x2 & (x3 & x4))) <-> x5`	0.012	0.537	0.642	0.901

3.4. Автоматические эксперименты

Результаты массового бенчмарка, сгруппированные для каждого числа входных переменных представлены в файле benchmark_results.csv и визуализированы графиком на рис. 3.1, где по оси абсцисс отложено число входных переменных, а по оси ординат (в логарифмическом масштабе) – среднее время работы методов в миллисекундах.

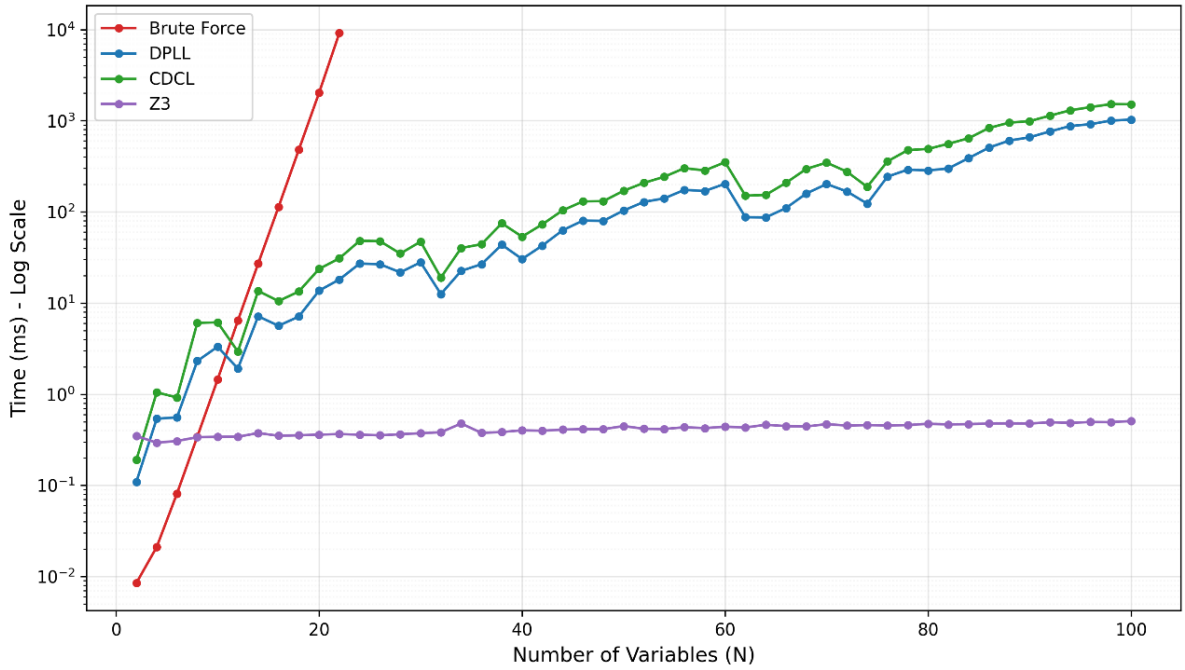

Рисунок 3.1 – Сравнительный график времени выполнения

Кривая bruteforce демонстрирует ярко выраженный экспоненциальный рост. На интервале от 2 до 20 переменных время увеличивается примерно на два порядка (от долей миллисекунды до единиц миллисекунд). При $n \approx 25$ время достигает нескольких десятков миллисекунд, и при $n > 25$ функция автоматически прерывается из-за установленного ограничения. Этот результат согласуется с теоретической оценкой $O(2^n)$.

В отличие от полного перебора, кривая Z3 остаётся практически горизонтальной на всём диапазоне, варьируясь в пределах 0.3–0.8 мс независимо от числа переменных. Это демонстрирует высокую масштабируемость современного SAT-решателя. Даже на сложных формулах с 100 переменными время работы остаётся на уровне долей миллисекунды.

Кривые DPLL и CDCL занимают промежуточное положение между Z3 и полным перебором. Их временная сложность растёт медленнее, чем в брутфорсе, но быстрее, чем у Z3. На малых значениях n (до 10 переменных) все три метода показывают сопоставимое время. При увеличении n кривые DPLL и CDCL растут и отличаются на константное время. При этом DPLL работает быстрее CDCL. Однако обе учебные реализации остаются значительно медленнее промышленного Z3.

3.5. Выводы по третьей главе

Сопоставление данных табл. 3.1 и 3.2, а также поведения кривых на рис. 3.1 позволяет сделать несколько важных выводов: во-первых, все подходы дают совпадающие логические результаты на широком наборе тестов, что подтверждает корректность реализации DPLL, CDCL и отображения в язык Z3; во-вторых, метод полного перебора демонстрирует ожидаемый экспоненциальный рост времени работы и уже в области порядка двух десятков входных переменных выходит за пределы интерактивного использования, тогда как SAT-решатель остаётся практически инвариантным к увеличению числа переменных в рассматриваемом диапазоне.

Таким образом, экспериментальное исследование подтверждает теоретические оценки из предыдущих глав и показывает, что при верификации реальных комбинационных схем, содержащих десятки и более входов, использование SAT-решателей на основе митора является не просто удобной, а фактически необходимой технологией, позволяющей автоматически доказывать эквивалентность или находить контрпримеры там, где прямой перебор всех входных векторов становится вычислительно недостижимым.

ЗАКЛЮЧЕНИЕ

В выполненной работе исследована задача проверки эквивалентности комбинационных логических схем с использованием SAT-решателей и традиционного метода полного перебора входных векторов. На основе теоретического анализа сформулирована связь между структурным описанием схем и их представлением в виде булевых формул, показано использование конструкции митора для сведения проверки эквивалентности к задаче выполнимости булевой формулы и обоснован выбор промышленного решателя Z3 в качестве основного инструмента SAT-проверки.

На базе полученных теоретических результатов разработан программный комплекс на языке Python, включающий единый модуль разбора булевых формул (лексический и синтаксический анализ, построение абстрактного синтаксического дерева) и независимые процедуры проверки эквивалентности. Реализованы вспомогательные скрипты для интерактивного тестирования, автоматического бенчмаркинга и визуализации результатов, что позволило не только продемонстрировать корректность работы алгоритмов на отдельных примерах, но и провести систематическое сравнение их производительности.

Экспериментальное исследование показало, что для схем с малым числом входных переменных оба подхода дают сопоставимое время работы и совпадающие логические результаты, что подтверждает корректность реализации парсера, построения митора и интерфейса к Z3. При увеличении числа входов метод полного перебора демонстрирует ожидаемый экспоненциальный рост времени (до десятков и сотен миллисекунд уже при 20–25 входах), тогда как среднее время работы SAT-подхода на базе Z3 остаётся практически постоянным в узком интервале долей миллисекунды на всём исследованном диапазоне размеров задач, а среднее время DPLL и CDCL растёт с небольшой скоростью, что позволяет использовать их на схемах с 25+ входами.

Полученные результаты позволяют сделать вывод, что использование SAT-решателей является практически необходимым инструментом при верификации эквивалентности комбинационных схем даже умеренной размерности, тогда как традиционный полный перебор быстро выходит за рамки приемлемого времени вычислений.

СПИСОК ЛИТЕРАТУРЫ

1. Handbook of satisfiability / ed. by A. Biere, M. J. H. Heule, H. van Maaren, T. Walsh. – Amsterdam: IOS Press, 2009.

2. Clarke E. M., Grumberg O., Peled D. Model checking. – Cambridge, MA: MIT Press, 1999.

3. Z3: SMT Solver: Documentation [Электронный ресурс] / Microsoft Research. – Режим доступа: https://github.com/Z3Prover/z3. – Дата обращения: 11.12.2025.