Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика и системы управления
КАФЕДРА Теоретическая информатика и компьютерные технологии

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

НА ТЕМУ:

“Фреймворк для автоматического анализа структуры многозначных логик”

Студент ИУ9-82(Б)	_____ (Подпись, дата)	Е. Д. Шевляков (И.О.Фамилия)
Руководитель ВКР	_____ (Подпись, дата)	А. Н. Непейвода (И.О.Фамилия)
Консультант	_____ (Подпись, дата)	_____ (И.О.Фамилия)
Консультант	_____ (Подпись, дата)	_____ (И.О.Фамилия)
Нормоконтролер	_____ (Подпись, дата)	_____ (И.О.Фамилия)

2021 г.

Аннотация

Объем данной составляет 50 страниц. Для ее написания были использованы 8 источников. В работе содержится 18 рисунков и 9 листингов.

Работа состоит из пяти частей. В первой части – обзор предметной области, во второй – ставится задача, в третьей – излагается разработка фреймворка, в четвертой – описывается ход тестирования, в пятой – предложено руководство для пользователя. В приложении приведены некоторые листинги.

Объектом исследования ВКР являются многозначные логики.

Цель работы – создание функционала, принимающего на вход набор логик и выдающего результат их сравнительного анализа в удобном пользователю виде.

В работе показано, что подобный программный продукт до этого не был реализован, и являлся бы полезным для исследователей в области логики.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

При изучении логик (множество которых велико), исследователю может понадобиться проверить некоторую гипотезу, например: функциональное вложение одной логики в другую, наличие лишней функции в базисе логики (при исключении которой из базиса, логика будет порождать все тот же класс функций) и тому подобные. В данной работе с этой целью рассматриваются конечнозначные логики.

Многозначная логика характеризуется тройкой $\{V, \mathcal{F}, D\}$, где V – множество логических значений, $\mathcal{F}$ – множество базисных логических операций (связок), $D \subset V$ – множество выделенных значений. Обычно, у многозначной логики есть представление также в виде систем вывода аксиом. Такое представление логики называется матричным. Помимо матричного представления, большинство многозначных логик описываются также как дедуктивные системы. Иногда, в этом случае оказывается, что разные дедуктивные системы описывают сходные логические системы, что бывает легче выяснить с помощью анализа ее матричного представления [3].

В 1973 году Сетте построил исчисление [3] P^1 с двумя связками (отрицание и импликация), правилом вывода modus ponens и следующими аксиомами:

1. $A \Rightarrow (B \Rightarrow A)$
2. $(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$
3. $(\neg \mathrm{A} \Rightarrow \neg \mathrm{B}) \Rightarrow ((\neg \mathrm{A} \Rightarrow \neg\neg \mathrm{B}) \Rightarrow \mathrm{A})$
4. $(A \Rightarrow B) \Rightarrow \neg\neg(A \Rightarrow B)$

В 1974 году да Коста построил для нее таблицы истинности в трехзначной семантике.

В 1995 году Сетте и Карниэлли описали «слабо интуиционистскую логику» [3] $\mathbf{I}^1$ с правилом вывода modus ponens и следующими аксиомами и определили, что она также выражается в виде трехзначной:

1. $A \Rightarrow (B \Rightarrow A)$
2. $(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$

3. $(\neg\neg A \Rightarrow \neg B) \Rightarrow ((\neg\neg A \Rightarrow B) \Rightarrow \neg A)$

4. $\neg\neg(A \Rightarrow B) \Rightarrow (A \Rightarrow B)$

Некоторое время логики $\mathbf{P}^1$ и $\mathbf{I}^1$ изучались независимо, пока в 2011 году не оказалось, что они функционально эквивалентны. Этот факт указывает на необходимость уметь решать задачу определения функциональных отношений между логиками.

При решении задачи построения функциональных отношений между логиками также возникает задача оценки существования супремума: если логика $\mathcal{L}$ функционально вкладывает $\mathcal{L}_1$ и $\mathcal{L}_2$, верно ли, что объединение их базисов порождает базис $\mathcal{L}$, или существует некоторая логика $\mathcal{L}'$, не совпадающая с $\mathcal{L}$, в которую вкладываются $\mathcal{L}_1$ и $\mathcal{L}_2$?

Ответ на этот вопрос важен при анализе структур предпорядков по отношению функционального вложения.

Проверки всех таких предположений зачастую могут потребовать непомерного количества времени в силу объемности вычислений. Вручную придется использовать эвристики и недопустимо большой перебор. Учитывая еще и человеческий фактор: при длительной однообразной работе увеличивается возможность допущения ошибки, такие задачи не разумно выполнять вручную.

Цель данной работы – создать инструмент, позволяющий автоматизировать такие проверки. Множество выделенных значений не используется, поскольку рассматриваются не доказательные свойства логик, а их функциональные свойства (т.е. их свойства как множества функций).

1 Обзор предметной области

1.1 Базовые определения

Согласно [2] и [3].

Пусть $\mathcal{F}m$ есть множество формул, построенных обычным рекурсивным способом из $Var = \{p_1, p_2, \ldots\}$ – конечного множества пропозициональных переменных и $Con = \{F_1, \ldots F_n\}$ – конечного множества пропозициональных связок, причем каждой связке F_i сопоставлено натуральное число $a(F_i)$, которое обозначает число ее аргументов. $\{\wedge, \vee, \rightarrow, \neg\} \subset Con$.

Литералы $\mathcal{F}m$ есть множество всех формул вида $\neg^k p$, где $\neg^0 p = p$ и $\neg^{k+1} p = \neg\left(\neg^k p\right)$, для $p \in Var$. Формулы, содержащие бинарные связки, назовем комплексными.

Алгебру формул $\mathcal{L} = \langle\ \mathrm{Fm},\ F_1, \ldots, F_m\rangle$ будем называть пропозициональным языком или $\mathcal{L}$-алгеброй.

1.2 Матричные логики

Согласно [2] и [3].

Пусть $A = \langle V, f_1, ..., f_m\rangle$– алгебра того же типа, что и пропозициональный язык $\mathcal{L}$, где V – множество истинностных значений и f_i – функция на V той же местности, что и F_i.

$\mathcal{L}$-матрица (логическая матрица) – это пара $\mathcal{A} = \langle A, F\rangle$, где A это $\mathcal{L}$-алгебра и F есть универсум из A, элементы из F называются выделенными элементами в $\mathcal{A}$.

Оценка в матрице $\mathcal{A}$ – это функция $v : Var \longrightarrow A$. Такая оценка v может быть рекурсивно расширена в $\bar{v} : \mathcal{F}m \longrightarrow A$ обычным способом. Также будем писать α^v для $\bar{v}(\alpha)$.

Для данной матрицы $\mathcal{A}$, определим отношение $=_{\mathcal{A}}$ тавтологии для формулы φ следующим образом: $\vDash_{\mathcal{A}} \varphi$ тогда и только тогда, когда для любой оценки v, $\varphi^v \in F$.

Формула φ логически следует из Γ – множества формул из $\mathcal{A}$ (сокращенно $\Gamma \vDash_{\mathcal{A}} \varphi$) $\Leftrightarrow$ не существует такой v в $\mathcal{A}$, что $\forall \psi^v \in F$: $\psi \in \Gamma$ $\varphi^v \in F$.

Для класса матриц $\mathbb{M}$ определим отношение $F_{\mathbb{M}}$ таким образом: $\Gamma \vDash_{\mathbb{M}} \varphi$ тогда и только тогда, когда $\Gamma \vDash_{\mathcal{A}} \varphi$, для всех $\mathcal{A} \in \mathbb{M}$.

Логику можно определить как дедуктивную систему $\mathcal{S}$ посредством аксиом (набором формул над пропозициональным языком) и правил вывода. Правило вывода в конечнозначных логиках подразумевает *modus ponens*: $\{A, A \to B\} \vDash_{\mathcal{A}} B$. Класс матриц $\mathbb{M}$ будем называть матрицами семантики $\mathcal{S}$ если для всех $\Gamma \cup \{\varphi\} \subseteq \mathcal{F}m, \Gamma \vdash_s \varphi$ тогда и только тогда, когда $\Gamma \vDash_{\mathbb{M}} \varphi$.

1.3 Литерально-паранепротиворечиво-параполные матрицы

Согласно [2] и [3].

Пусть A – такое множество логических значений, что $\{0, 1\} \subseteq A$ и $F \subseteq A$, причем $1 \in F$ и $0 \notin F$. Пусть функция $\sim : A \longrightarrow A$ удовлетворяет условиям $\sim 1 = 0$ and $\sim 0 = 1$. Определим литерально-паранепротиворечиво-параполные матрицы (или *LLP*-матрицы) как $\langle A, F, \sim \rangle$ со следующими операциями:

$$
\begin{aligned}
a \vee b &= \begin{cases} 1 & \text{if } a \in F \text{ or } b \in F \\ 0 & \text{otherwise} \end{cases} \\
a \wedge b &= \begin{cases} 1 & \text{if } a \in F \text{ and } b \in F \\ 0 & \text{otherwise} \end{cases} \\
a \to b &= \begin{cases} 1 & \text{if } a \notin F \text{ or } b \in F \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
\tag{1}
$$

Заметим, что LLP-матрица $\langle A, F, \sim \rangle$ – это просто матрица в обычном понимании, где алгебра над универсумом A имеет 3 бинарных операции, определенных выше, и одну унарную $\sim$.

LPP-матрица является паранепротиворечивой, если для некоторого $a \in A$ выполняется $a \in F$ и $\sim a \in F$; является параполной, если $a \notin F$ и $\sim a \notin F$. Обратим внимание, что название литерально-паранепротиворечиво-параполный может ввести в заблуждение, поскольку не все такие матрицы являются паранепротиворечивыми и параполными. На самом деле, матрица $\langle \{1, 0\}, \{1\}, \sim \rangle$, где $\sim 0 = 1$ и $\sim 1 = 0$, которая определена в классической логике, не является ни паранепротиворечивой, ни параполной.

Левин и Микенберг определяют непротиворечивую и полную дедуктивную систему для логики, задаваемой классом всех LPP-матриц $\langle F, A, \sim \rangle$, не налагая условия на *F, A* или $\sim$. Эта система названа литеральная паранепротиворечивая параполная логика ($LPPL$) с единственным правилом вывода *modus ponens* и со следующими аксиомами:

(A1) $\alpha \to (\beta \to \alpha)$

(A2) $(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$

(A3) $(\alpha \wedge \beta) \to \alpha$

(A4) $(\alpha \wedge \beta) \to \beta$

(A5) $(\alpha \to \beta) \to ((\alpha \to \gamma) \to (\alpha \to (\beta \wedge \gamma)))$

(A6) $\alpha \to (\alpha \vee \beta)$

(A7) $\beta \to (\alpha \vee \beta)$

(A8) $(\alpha \to \gamma) \to ((\beta \to \gamma) \to ((\alpha \wedge \beta) \to \gamma))$

(A9) Аксиома отрицания : $(\sim A \to \sim B) \to (B \to A)$, где *A* и *B* – сложные формулы.

1.3.1 Паралогики Левина-Микенберг

В данной работе в качестве объекта исследований рассмотрен класс четырехзначных LPP-логик Левина-Микенберг. Мы рассматриваем их как

пространства функций над четырехэлементным множеством {0, ⊥, ⊤, 1}. Здесь 1 и 0 – обычные логические значения "истина" и "ложь", а ⊥ и ⊤ - дополнительные. Над этим множеством заданы четыре набора операций ∧, ∨, →.

1. Первый набор (условно обозначаемый O_1). Здесь со значениями ⊤ и ⊥ логические операции обращаются как с 0. Набор на рисунке 1.

$\vee_1$	1	⊤	⊥	0
1	1	1	1	1
⊤	1	0	0	0
⊥	1	0	0	0
0	1	0	0	0

$\wedge_1$	1	⊤	⊥	0
1	1	0	0	0
⊤	0	0	0	0
⊥	0	0	0	0
0	0	0	0	0

$\rightarrow_1$	1	⊤	⊥	0
1	1	0	0	0
⊤	1	1	1	1
⊥	1	1	1	1
0	1	1	1	1

Рисунок 1. Первый набор операций

2. Второй набор (условно обозначаемый O_2). Здесь со значениями ⊤ и ⊥ логические операции обращаются как с 1. Набор на рисунке 2.

$\vee_2$	1	⊤	⊥	0
1	1	1	1	1
⊤	1	0	0	0
⊥	1	0	0	0
0	1	0	0	0

$\wedge_2$	1	⊤	⊥	0
1	1	0	0	0
⊤	0	0	0	0
⊥	0	0	0	0
0	0	0	0	0

$\rightarrow_2$	1	⊤	⊥	0
1	1	0	0	0
⊤	1	1	1	1
⊥	1	1	1	1
0	1	1	1	1

Рисунок 2. Второй набор операций

3. Третий набор (условно обозначаемый O_3). Здесь со значением ⊤ операции обращаются как с 1, а с ⊥ как с 0. Набор на рисунке 3.

$\vee_3$	1	⊤	⊥	0
1	1	1	1	1
⊤	1	0	0	0
⊥	1	0	0	0
0	1	0	0	0

$\wedge_3$	1	⊤	⊥	0
1	1	0	0	0
⊤	0	0	0	0
⊥	0	0	0	0
0	0	0	0	0

$\rightarrow_3$	1	⊤	⊥	0
1	1	0	0	0
⊤	1	1	1	1
⊥	1	1	1	1
0	1	1	1	1

Рисунок 3. Третий набор операций

4. Четвертый набор (условно обозначаемый O_4). Здесь со значением $\top$ операции обращаются как с 0, а с $\bot$ как с 1. Набор на рисунке 4.

$\vee_4$	1	$\top$	$\bot$	0
1	1	1	1	1
$\top$	1	0	0	0
$\bot$	1	0	0	0
0	1	0	0	0

$\wedge_4$	1	$\top$	$\bot$	0
1	1	0	0	0
$\top$	0	0	0	0
$\bot$	0	0	0	0
0	0	0	0	0

$\rightarrow_4$	1	$\top$	$\bot$	0
1	1	0	0	0
$\top$	1	1	1	1
$\bot$	1	1	1	1
0	1	1	1	1

Рисунок 4. Четвертый набор операций

Кроме того, рассматриваются 16 разных отрицаний. Операции на рисунке 5.

	$\neg_1$	$\neg_2$	$\neg_3$	$\neg_4$	$\neg_5$	$\neg_6$	$\neg_7$	$\neg_8$	$\neg_9$	$\neg_{10}$	$\neg_{11}$	$\neg_{12}$	$\neg_{13}$	$\neg_{14}$	$\neg_{15}$	$\neg_{16}$
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$\top$	1	0	0	1	$\top$	$\bot$	$\bot$	0	$\top$	$\top$	1	0	$\top$	$\bot$	$\bot$	1
$\bot$	1	0	1	0	$\bot$	$\top$	1	$\top$	1	0	$\bot$	$\bot$	$\top$	$\bot$	0	$\top$
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Рисунок 5. Набор отрицаний

Отметим, что данные наборы операций удовлетворяют всем необходимым условиям LPP-матриц, введенным в пункте 1.3.

Далее обозначаем четырехзначную логику Левина-Микенберг с операциями O_i, $\neg_j$ как *LM(i,j)*.

Изначально уже имелся результат, описывающий структуру функциональных вложений (см. 1.4) для 16-ти логик Левина-Микенберг. Полурешетка для данных логик представлена на рисунке 6.

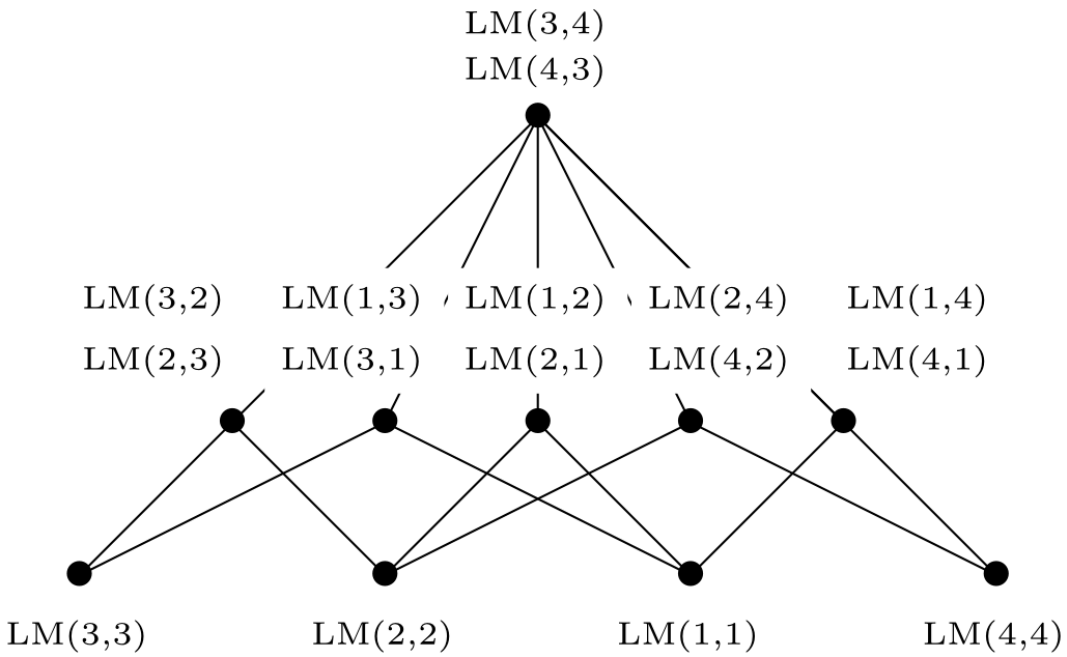

Рисунок 6. Полурешетка функциональных вложений для 16 логик Левина-Микенберг

Заметим, что данная структура обладает свойствами верхней полурешетки (см. 1.5): у каждых двух логик есть точная верхняя грань по отношению функционального вложения.

Для 64 логик задача была решена только для некоторых частных случаев.

1.4 Функциональное вложение

В данной задаче не рассматриваются дедуктивные свойства логик. Поэтому далее по умолчанию под логикой будем иметь в виду алгебру $\langle F, V \rangle$, где F – пропозициональный язык с функцией оценки, V – множество логических значений.

Пусть $L_1 = \langle F_1, V \rangle$, $L_2 = \langle F_2, V \rangle$.

Операция суперпозиции состоит из элементарных операций над элементами множества F_1, таких как переименование и отождествление переменных и подстановка одной функции в другую.

Замыканием $[F_1]$ будем называть минимальное множество функций, которое содержит все суперпозиции функций, принадлежащих F_1.

Логика L_1 функционально вложима в логику $L_2 \Leftrightarrow \forall F_1 \in [F_2]$.

Логика L_1 функционально эквивалентна логике L_2, если и только если выполняются следующие два условия:

1. Логика L_1 функционально вложима в логику L_2 .
2. Логика L_2 функционально вложима в логику L_1 .

1.5 Предпорядок, решетка и полурешетка

Предпорядком на множестве M называется бинарное отношение, обладающее свойствами рефлексивности и транзитивности [4]:

$$\forall a \in M : a \leqslant a$$
$$\forall a, b, c \in M : (a \leqslant b \wedge b \leqslant c) \Rightarrow (a \leqslant c) \tag{2}$$

Алгебраическая решетка [1] – множество с тремя операциями $\langle L, \wedge, \vee, \neg \rangle$, где L – непустое множество элементов решетки. При этом выполняются следующие требования:

1. $a \wedge b = b \wedge a, \quad a \vee b = b \vee a.$
2. $a \wedge (b \wedge c) = (a \wedge b) \wedge c, \quad a \vee (b \vee c) = (a \vee b) \vee c.$
3. $(a \wedge b) \vee b = b, \quad (a \vee b) \wedge b = b.$
4. $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c), \quad a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c).$
5. $(a \vee \neg a) \wedge b = b, \quad (a \wedge \neg a) \vee b = b.$

Алгебраическая полурешетка [4] – множество с одной операцией $\langle L, \vee \rangle$. L – непустое множество элементов решетки. При этом выполняются следующие требования:

1. $a \vee a = a.$
2. $a \vee b = b \vee a.$
3. $(a \vee b) \vee c = a \vee (b \vee c).$

Супремумом подмножества в предупорядоченном множестве называется наименьший элемент, который не меньше каждого элемента этого подмножества.

Инфинумом подмножества в предупорядоченном множестве называется наибольший элемент, который не больше каждого элемента этого подмножества.

Супремум и инфинум согласованы с алгебраической решеткой: $\wedge$ – супремум, $\vee$ – инфинум.

Полурешетки бывают верхними и нижними. В верхних рассматривается операция взятия верхней грани, в нижних – наоборот.

1.6 Диаграммы Хассе

Поскольку предпорядок является бинарным отношением, он может быть представлен ориентированным графом. Предпорядок, заданный функциональным вложением, является рефлексивным, транзитивным и антисимметричным, т.е. частничным порядком. Это позволяет упростить графическое представление частично упорядоченного множества, выполнив следующие шаги:

- Удалить все дуги в себя.
- Удалить все транзитивные ребра.
- Удалить направления по краям, предполагая, что они ориентированы вверх. Итак, если a⩽b, тогда вершина b появляется над вершиной a.

Полученный график хорош своей наглядностью и называется диаграммой Хассе.

В качестве примера рассмотрим соотношение делимости $a \mid b$ на множестве $A = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. Ориентированный граф и диаграмма Хассе данного соотношения приведены на рисунке 7.

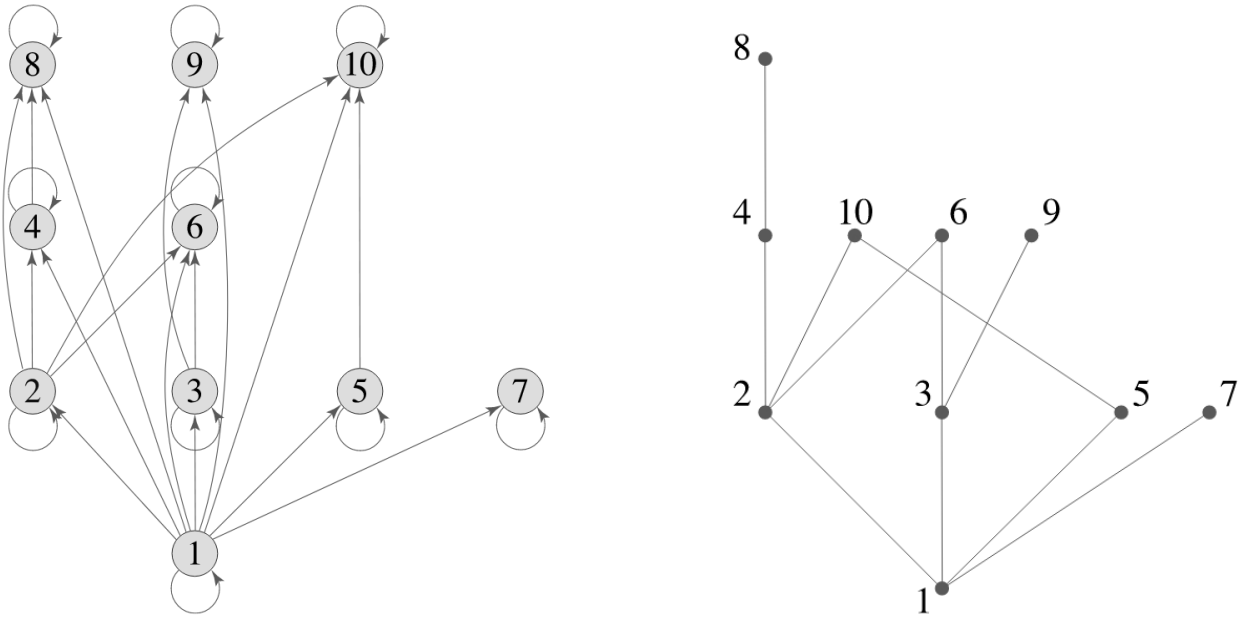

Рисунок 7. Орграф (слева), диаграмма Хассе (справа)

1.7 Синтаксис рефала

РЕФАЛ (Рекурсивных Функций Алгоритмический язык) [6] – это функциональный язык программирования, ориентированный на так называемые "символьные преобразования": обработку символьных строк (например, алгебраические выкладки), перевод с одного языка (искусственного или естественного) на другой, решение проблем, связанных с искусственным интеллектом.

Программа, написанная на Рефале, состоит из функций. Каждая функция, в свою очередь, состоит из заглавия и тела, в котором описан ряд предложений в фигурных скобках. Под предложением понимается пара (образец, выражение), разделенная знаком равенства: =.

Работа функции заключается в сопоставлении ее аргумента (у функций Рефала всегда один аргумент) с образцами: сначала аргумент сопоставляется с образцом первого предложения. Если сопоставление дало положительный результат, то вычисляется выражение этого предложения (значение, полученное

в результате вычисления выражения, является итогом работы функции). Если же сопоставление прошло неудачно, то те же самые действия проходят на следующем предложении, и так далее. Если аргумент не удалось сопоставить ни с каким образцом из предложений функции, то возникает ошибка применения функции. Пример функции Рефала представлен на листинге 1.

```
Palindrom {
      s.1 e.2 s.1 = <Palindrom e.2>;
      s.1 = True;
      /* пусто */ = True;
      e.1 = False;
}
```

Листинг 1. Пример функции на Рефале.

Как видно на листинге, образец представляет собой некую последовательность, элементы которой разделены пробельными символами, незаключенными в кавычки.

Рассмотрим все типы элементов, из которых могут состоять образцы:

- Символы – последовательность знаков, которая не распознается как переменная.
- Переменный символ – Обозначается как *s.x*, где *x* есть некая последовательность букв, цифр или знаков. Переменная этого типа может принимать значение любого символа.
- Терм – Обозначается как как t.x, где x есть некая последовательность букв, цифр или знаков. Терм может принимать значение символа либо объектного выражения, заключенного в структурные скобки. Объектное выражение есть выражение, состоящее из символов и структурных скобок. Скобки должны образовывать правильную скобочную структуру. Выражение может быть, в частности, и пустым. Знаки "пробел", не входящие в

последовательность простых символов, служат для внешнего оформления программы.

- Переменные типа *e.x* – отвечают последовательности термов (в том числе и пустой).

Использование s-переменных и сопоставления сверху вниз позволяет сильно сэкономить место при определении конечнозначных логических связок.

2 Постановка задачи

2.1 Описание задачи

Многозначная логика характеризуется тройкой $\{V, \mathcal{F}, D\}$, где V – множество логических значений, $\mathcal{F}$ – множество базисных логических операций (связок), $D \subset V$ – множество выделенных значений [3].

Формула считается тавтологией (теоремой), если для всех значений аргументов ее значение принадлежит D. В данной работе множество выделенных значений не используется, поскольку рассматриваются не доказательные свойства логик, а их функциональные свойства (т.е. их свойства как множества функций).

Требуется построить инструмент, позволяющий определять функциональное вложение пространств конечнозначных функций, которые мы также называем логиками. Инструмент должен обеспечивать возможность компактного описания конечнозначных функций и вывод результата как в текстовом виде, так и в виде диаграммы Хассе.

Данный инструмент может быть использован в следующих целях:

1. Проверка выразимости одних связок через другие в конечнозначной логике, что позволит выявлять базисные наборы операций;
2. Проверка выразимости базиса одной логики в другой логике с теми же истинностными значениями, что позволит устанавливать функциональную эквивалентность и функциональное вложение.
3. Проверка выразимости базиса логики в объединенном базисе нескольких других логик – это позволяет проверять, обладают ли предпорядки на множествах логик, заданные отношением функционального вложения, свойством супремума.

2.2 Входные данные

Программа читает описания многозначных логик из файлов с расширениями *mvlog*, определенных следующим образом:

Name = имя структуры

Values = [список логических значений через запятую]

Список логических функций (унарных или бинарных) в рефал-синтаксисе

Структура логической функции:

имя {

Список предложений

}

Структура предложения:

образец = результат;

Результат есть всегда единственное логическое значение (должно присутствовать в списке *Values*). Образец, определяющий бинарную функцию - это два терма через пробел, каждый из которых является либо логическим значением, либо символьной переменной (s-переменной).

Предложения в описании унарной функции отличаются тем, что в образцах есть ровно один терм.

Фреймворк получает на вход список как минимум из двух файлов с описанием многозначных логик. Пример содержания входного файла приведен на листинге 2.

```
Name = L2
Values = [0, 1]

and {
 1 1 = 1;
 s.x s.y = 0;
}

or {
 0 0 = 0;
 s.x s.y = 1;
}

imply {
   1 0 = 0;
   s.x s.y = 1;
}

not {
   1 = 0;
   0 = 1;
}
```

Листинг 2. Пример входного файла.

Повторная *s*-переменная также допустима. В этом случае фреймворк, при заполнении матрицы логической функции, будет перебирать пары одинаковых логических значений.

2.3 Выходные данные

Фреймворк записывает результаты сравнения каждых двух логик, определяемых входными файлами, в виде списка следующих предложений:

Имя логики-i and Имя логики-j: результат

Результат – либо значение *embedded*, либо значение *embeds*, либо значение *equivalent*, либо значение *non-comparable*. Первое выдается, если множество функций, определенное первой логикой, шире, чем множество функций второй; второе – если наоборот; третье – если эти множества совпадают; и четвертое – если не выполняется ни одно из вышеперечисленных.

Возможно использовать другие форматы выходных данных. Важно лишь то, чтобы извлекалась необходимая для анализа информация.

2.4 Описание используемых оптимизаций

Актуальной задачей для увеличения скорости перебора является доказательство невозможности функционального вложения, поскольку в случае использования полного перебора для него придется порождать полное функциональное описание логики, тогда как для доказательства вложения обычно хватает намного меньшей глубины поиска.

Оптимизацией будем называть некоторую лемму о невозможности определимости функции в модели многозначной логики.

Хотя использование лемм существенно сокращает объем перебора, все равно в случае, когда нужно определить структуру вложений большого количества логик, этот перебор может остаться слишком большим для осуществления человеком вручную. Мы проверили, что использование подобных лемм может быть эффективно реализовано программно.

Рассмотрим некоторые рассуждения, приведенные в [3]. Матрица логики P^{2*} имеет следующий вид:

$$\mathcal{A}^{P^{2*}} = \langle \{0, \bot, \top, 1\}, \neg_{16}, \to_{P^2}, \{1, \top, \bot\} \rangle$$

Описание импликации $\to_{P^2}$ приведено на рисунке 8.

$\to_{P^2}$	1	$\top$	$\bot$	0
1	1	1	1	0
$\top$	1	1	$\top$	0
$\bot$	1	$\top$	1	0
0	1	1	1	1

Рисунок 8. Описание импликации $\to_{P^2}$

Было приведено следующее доказательство, что посредством связок P^{2*} невозможно выразить отрицание $\neg_7$. Поскольку функции, соответствующие связкам P^{2*}, принимают значения из множества $\{0, \top, 1\}$, в то время как существует такое v, что $\neg v = \bot$.

Данное доказательство соответствует использованию оптимизации Область значений 1.

Рассмотрим также следующее рассуждение из [3]. Показано, что посредством связок $\{\to_2, \neg_{16}\}$ нельзя выразить связку $\neg_8$, поскольку любое применение связки $\to_2$ породит значение из $\{0, 1\}$, а это множество сохраняется связками $\{\to_2, \neg_{16}\}$. Поэтому единственный способ выразить $\neg_8$ через них – это применять только связку $\neg_{16}$. Повторное ее применение порождает функцию в $\{0, 1\}$, что доказывает, что выразимости нет. Данное доказательство соответствует использованию оптимизации Область значений 2.

Рассмотрим утверждения о свойствах логических операций, используемых в разработанном инструменте с целью оптимизации перебора.

Коммутативность

Если $\odot$ коммутативна, то $A \odot B$ неопределима через множество $F \Leftrightarrow B \odot A$ неопределима через множество F.

Если матрица некоторого бинарного оператора $\odot$ симметрична, тогда нет необходимости порождать две новые функции $A \odot B$ и $B \odot A$, достаточно одной.

Неразличимость значений

V – множество логических значений

R_L – отношение неразличимости для логики L. Данное отношение определено следующим образом:

$$v_1 R_L v_2 \Rightarrow \forall f \in L \quad \forall x_1, ..., x_n \in V$$
$$f(x_1, ..., x_{i-1}, v_1, x_{i+1}, ..., x_n) = f(x_1, ..., x_{i-1}, v_2, x_{i+1}, ..., x_n) \quad (3)$$

Оптимизация использует следующее утверждение:

$$\exists v_1, v_2 \in V : \langle v_1, v_2 \rangle \in R_{L_1} \wedge \langle v_1, v_2 \rangle \notin R_{L_2} \Longrightarrow L_2 \nsubseteq L_1 \quad (4)$$

В логике L_1 находятся все пары неразличимых элементов. Если какая-либо из этих пар отсутствует в парах неразличимых элементов логики L_2, то L_1 не может функционально вкладывать L_2.

В следующих двух оптимизациях используется понятие множества значений, сохраняемого функцией. Говорят, что функция f сохраняет значения из множества W, если $\forall x_1, .., x_n \in W \quad f(x_1, ..., x_n) \in W$ [5].

Область значений 1

Данная оптимизация использует следующее утверждение:

$$\exists W \subseteq V : \forall x_1, ..., x_n \in W, \forall f \in L_1 : f(x_1, ..., x_n) \in W \quad (5)$$
$$\exists g \in L_2 : \exists x_1, ..., x_n \in W : g(x_1, ..., x_n) \notin W \Longrightarrow L_2 \nsubseteq L_1$$

Пусть задача состоит в том, чтобы проверить, выразим ли оператор g через множество операторов f_i. Пусть множество всех логических значений, на которых действуют операторы g, f_i – это V. Если существует такое подмножество W множества V, что для всех $a, b \in W \quad f_i(a, b) \in W$, но для некоторых $a', b' \in W \quad g(a', b') \notin W$, тогда оператор g заведомо не выразим через f_i .

Пример: оператор, возвращающий всегда $\perp$, не определим ни в одной из паралогик $LM(i,j)$. Потому что все ператоры этих паралогик сохраняют значения из {0, 1}.

Область значений 2

V – множество логических значений.

Проверяем $L_2 \subseteq L_1$.

$$\exists W \subset V : \forall f \in L_1 \forall x_1, ..., x_n \in W \, f(x_1, ..., x_n) \in W \quad (6)$$

$\forall g \in L_2 : \mathrm{rng}(g) \supset W$ следует выражать из $\forall f \in L_1 : \mathrm{rng}(f) \supset W$

Пример: проверка вложения $LM(1,5)$ в $LM(1,6)$. Здесь условию области значений W={0, 1} удовлетворяют функции $\wedge_1, \vee_1, \to_1$, не удовлетворяет $\neg_6$. Зато $\neg_6$ сохраняет множество {0, 1}. Поэтому пытаться выразить $\neg_5$ в $LM(1,6)$ мы можем, используя только один оператор $\neg_6$. Весь перебор в этом случае пройдет в три шага, и сразу же будет получено доказательство того, что *LM(1,5)* в *LM(1,6)* не вкладывается функционально.

3 Разработка

Для решения задачи необходимо было реализовать:

- Парсер, который будет извлекать из входных файлов информацию и преобразовывать ее к подходящему для анализа виду.
- Алгоритм, строящий все возможные формулы с помощью имеющихся функций.
- Оптимизации, использующие свойства логик и здравый смысл для сокращения генерации формул в алгоритме.
- Функционал, сохраняющий результаты работы программы в удобном пользователю виде.

3.1 Парсер

3.1.1 Структурирование mvlog файла

На этом этапе из *mvlog* файла создается структура.

Разбор информации из входных файлов происходит с помощью регулярных выражений:

- `Name *= *[A-Za-z0-9_,\(\)\+]+\n` – строка с именем логики
- `Values *= *\[(\S+, *)*\S+\]\n` – строка со списком термов
- `[^\s,]+` – терм в строке термов
- `[A-Za-z0-9_]+ *\\{[^\\}]+\\}` – функция
- `[^\n]+=[^;]+` – предложение

В качестве представления информации из входного файла в виде структуры выступает класс *Mvlog*. Он имеет атрибуты:

- *name* – для хранения названия
- *values* – для хранения логических значений

- *functions* – для хранения функций

Атрибуты назначает метод *_parse* при создании экземпляра класса. *_parse* записывает все содержимое файла в переменную, затем, с помощью приведенных выше регулярных выражений, обнаруживает строку с именем и строку с термами, извлекает их. Если что-либо из этого не нашлось – порождает исключение. Далее находятся функции. Для их обработки используется класс *Function*. Аналогично классу *Mvlog*, *Function* имеет метод *_parse*, который получает текстовое представление функции, обнаруживает с помощью регулярных выражений в нем ее имя, и предложения. При обнаружении ошибок в предложениях порождает исключение.

3.1.2 Преобразование структуры к удобному для анализа виду

На этом этапе из созданной структуры получается сущность логики, предназначенная для взаимодействий с ней при переборе.

Для этой цели был разработан класс *Logic*. Он имеет атрибуты:

- *name* – имя логики
- *values* – термы логики
- *functions* – функции логики
- *eq_vals* – необходим для оптимизаций.

Атрибуты name и values переходят от объекта класса *Mvlog*, *functions* заполняется с помощью метода *_parse*. Рассмотрим, как именно:

Из каждой функции объекта класса *Mvlog* создается объект класса *TableFunction*. Этот класс воплощает операцию логики. У *TableFunction* есть:

- *name* – имя
- *values* – логические значения
- *dim* – мерность
- *data* – таблица значений
- *is_symmetric* – свойство коммутативности
- *value_area* – область значений

На этом этапе, по очереди просматриваются все предложения для каждой функции, в соответствии с предложением, заполняются ячейки таблицы. Если в предложении присутствует s-символ, то заполняются все пустые ячейки, удовлетворяющие оставшейся части предложения без s-символов. При вызове получившейся функции аргументам сопоставляются индексы, по которым идет обращение к ячейке таблицы. Повторная s-переменная также допустима. В этом случае фреймворк, при заполнении матрицы логической функции, будет перебирать пары одинаковых логических значений.

Таким образом из входных *mvlog* файлов получаются экземпляры класса *Logic*, функции которых есть экземпляры класса *TableFunction*.

3.2 Переборный алгоритм

Алгоритм призван ответить на вопрос, возможно ли из интересующего нас набора функций вывести набор других функций. В этих целях производится перебор.

Сначала выполняется некоторая подготовка, затем происходит цикл в котором есть две фазы: Развертка и Свертка. В развертке генерируются новые ветви развития формул из уже имеющихся (у каждого рассмотренного далее алгоритма будет свой набор действий для ветвления). В свертке закрываются ветви, не приводящие к новым результатам, остаются только перспективные. Когда все ветви будут закрыты, цикл завершается, и вернется результат: либо найдены нужные формулы, либо нет.

Было разработано 3 переборных алгоритма. Использовался объектно-ориентированный подход в целях удобства и наглядности. Диаграмма классов алгоритмов продемонстрирована на рисунке 9.

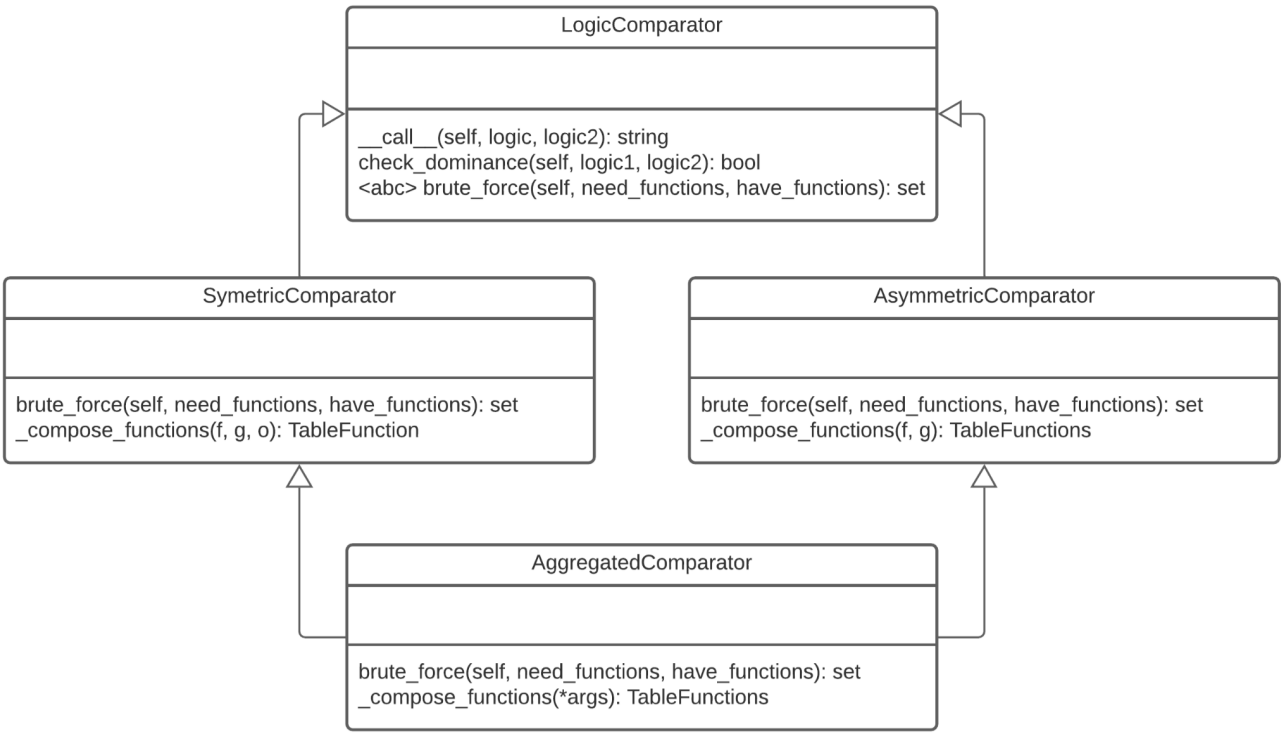

Рисунок 9. Диаграмма классов переборных алгоритмов

Класс *LogicComparator* наследуется от абстрактного. В нем определен метод *check_dominance*, который принимает две логики, сравнивает их с помощью метода *brute_force* (который как раз и нужно реализовать в наследниках) и возвращает результат. Класс *LogicComparator* приведен на листинге 3.

```
class LogicComparator(ABC):
    def __call__(self, logic1, logic2):
        dominance1 = self.check_dominance(logic1, logic2)
        dominance2 = self.check_dominance(logic2, logic1)

        if dominance1 and dominance2:
            return 'equivalent'
        elif dominance1:
            return 'embedded'
        elif dominance2:
            return 'embeds'
        else:
            return 'non-comparable'

    def check_dominance(self, logic1, logic2):
        ...

    @abstractmethod
    def brute_force(self, need_functions, have_functions):
        pass
```

Листинг 3. *LogicComparator*

Остальные классы и являются тремя алгоритмами перебора. Ключевую роль в них играет метод *_compose_function*, которым определяется способ появления новых функций. Метод *brute_force* реализует корректную последовательность применений метода *_compose_functions*. Рассмотрим каждый из алгоритмов.

Алгоритм *SymmetricComparator*

Список уже проанализированных операций *CheckedList* и список проверяемых функций *ListToCheck* составляем из матриц, наполненными каждым из логических значений из области значений логики и двух матриц, соответствующих пропозициональным переменным (одна с одинаковыми столбцами, другая с одинаковыми строками).

Циклически, пока *ListToCheck* не пуст, происходит следующий ряд действий с каждым элементом K из *ListToCheck*:

1. создаем операторы, применяя к K унарные операции логики.
2. создаем операторы вида $K \odot P$ и $P \odot K$ для каждой бинарной функции $\odot$ из базиса логики и каждого элемента P из *CheckedList*.
3. составляем новый *ListToCheck* из новых полученных операторов и также добавляем их в *CheckedList*.

При полном переборе, алгоритм пройдется по всем именам функций, которые возможно получить из анализируемой логики. Пример фрагмента семантической структуры функций с помощью *SymmetricComparator* представлен на рисунке 10.

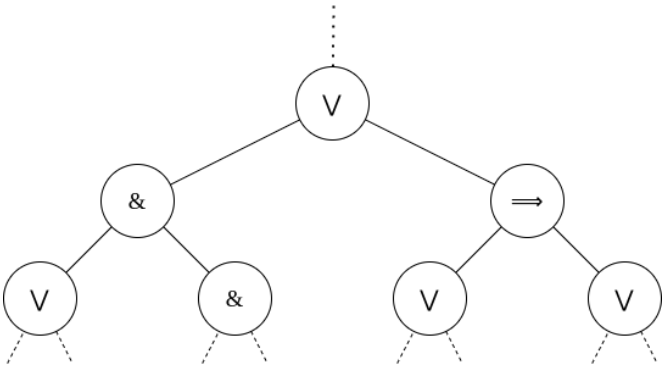

Рисунок 10. Дерево развития с *SymmetricComparator*

Алгоритм *AsymmetricComparator*

Здесь композиции двух операторов аналогичны применению унарного оператора, заданного данным бинарным.

ListToCheck и *CheckedList* наполняются базовыми функциями логики.

Циклически, пока *ListToCheck* не пуст, происходит следующий ряд действий с каждым элементом *K* из *ListToCheck* и каждым элементом *P* из *CheckedList*:

1. создаем операторы композициями вида *K(P)* и *P(K)*.
2. составляем новый *ListToCheck* из новых полученных операторов и также добавляем их в *CheckedList*.

В данном алгоритме, в отличие от *SymmetricComparator*, за один шаг комбинируются максимум две различные функции, в то время, как ином, таких может быть три. В связи с этим, алгоритм рассматривает не все функции.

С другой стороны, *AsymmetricComparator* показывает гораздо лучшие результаты по скорости, чем *SymmetricComparator*. Пример фрагмента семантической структуры функций с помощью *AsymmetricComparator* представлен на рисунке 11.

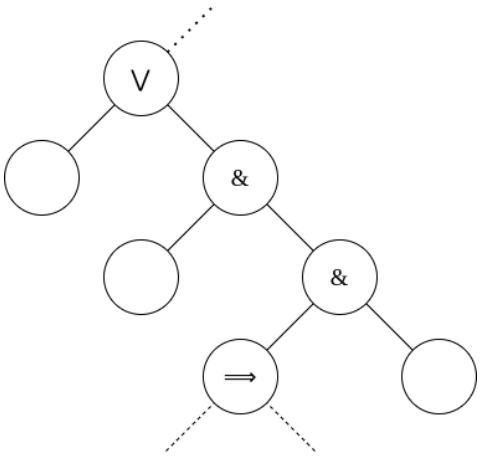

Рисунок 11. Дерево развития с *AsymmetricComparator*

Таким образом *SymmetricComparator* строит полное бинарное дерево развития функций, в то время как дерево *AsymmetricComparator* является списком, каждый элемент которого дополнен еще одной формулой.

Хотя *AsymmetricComparator* перебирает не все функции логики, но в случае, если функции обладают ассоциативностью и дистрибутивностью, сильно экономит время за счет того, что порождает меньше эквивалентных.

Поэтому при работе переборных алгоритмов естественно сначала воспользоваться им, и лишь при неудаче обращаться к полному перебору.

Алгоритм *AggregatedComparator*

Комбинирует идеи двух предыдущих алгоритмов. Рассмотрим *_compose_functions* у *AggregatedComparator*. Код этого полиморфного метода приведен на листинге 4.

```
@staticmethod
def _compose_functions(*args):
   if len(args) == 2:
       return AsymmetricComparator._compose_functions(*args)
   else:
       return SymmetricComparator._compose_functions(*args)
```

Листинг 4. Метод *_compose_functions* у *AggregatedComparator*

Таким образом, *_compose_functions* можно использовать как для быстрой генерации, так и для полной. На каждом обороте цикла строит операторы аналогично *AsymmetricComparator*, затем строит операторы аналогично *SymmetricComparator*. По скоростным показателям занимает второе место из трех.

3.3 Оптимизации

Алгоритмы выше не используют никакие свойства логических функций. С одной стороны, это позволяет применять их в целом при работе фреймворка,

а не только для конкретного класса логик, с другой стороны, делает их неэффективными.

Принимая во внимание свойства логических функций и руководствуясь логикой, можно сократить перебор функций (см. раздел 2.4). В процессе перебора иногда возможно предупредить генерацию уже существующей функции или пропустить перебор вообще, поняв, что он не обязателен для получения ответа.

Коммутативность

Если *TableFunction* имеет симметричную матрицу, то генерируется *f(a, b)*, а *f(a, b)* не генерируется.

Своевременная остановка

Если необходимые функции уже были получены, то нет необходимости завершать оборот цикла.

Во всех алгоритмах перебора разумно часто проверяется наличие еще не найденных функций.

Транзитивность

$$Ł_1 \subseteq L_2 \wedge L_2 \subseteq L_3 \Rightarrow L_1 \subseteq L_3 \tag{7}$$

Так, если мы уже получили, что L_1 *embedded/embeds/equivalent* L_2, L_2 *embedded/embeds/equivalent* L_3 соответственно, то при сравнении L_1 и L_3 ставим соответствующий результат и перебора не ведем.

Начиная с этой, все последующие оптимизации прописаны в модуле optimizations.py.

Для реализации транзитивности был создан класс *Transitivity*. В атрибуте *relations* хранится прямое произведение множества рассматриваемых логик на самого себя. При обращении к паре, возвращается строка с именем соответствующего отношения, если имя пусто, то отношения еще не было найдено.

Неразличимость значений

Для проверок такого условия в классе *Logic* предусмотрен атрибут *eq_vals*. В него записывается множество всех $\langle a, b\rangle : aR_Lb$. Сама проверка реализована в функции *indiscernible_of_values*. Если ложно:

$$\texttt{logic1.eq\_vals} \leqslant \texttt{logic2.eq\_vals}$$

то $\texttt{logic1} \not\simeq \texttt{logic2}$.

Вложение области отображения

$$\operatorname{rng}(L_1) \subset \operatorname{rng}(L_2) \Rightarrow L_1 \not\supseteq L_2 \tag{8}$$

Логика не может вкладывать другую, область отображения функции которой шире.

Для такой проверки у *TableFunction* предусмотрен атрибут *value_area*. У *Logic value_area* уже реализовано как свойство, собирается из всех *value_area* ее функций. Запускается эта проверка с помощью функции *embedding_of_value_area.* Если ложно:

$$\texttt{logic1.value\_area} \geqslant \texttt{logic2.value\_area}$$

то $\texttt{logic1} \not\simeq \texttt{logic2}$.

Область значений 1

Для проверки этой оптимизации была создана функция *closure_sets*. В ней находятся все W – подмножества V, сохраняемые в логике, и для каждого из них проверяется существование не сохраняющей его функции g.

Область значений 2

Для проверки этой оптимизации была создана функция *sets_w*. Ее сценарий таков:

1. Выбираются все подмножества логических значений, относительно которых замкнуты все функции `logic1`.

2. Для каждой функции `logic2` подбираются подходящие множества, полученные в пункте 1.
3. Для множеств, полученных в пункте 2, подбираются подходящие функции `logic1`.
4. Запускается перебор на выбранных функциях.
5. Возвращается набор функций, которые уже удалось найти.
6. Если функцию не удалось выразить полученным урезанным множеством функций иной логики, то возвращается *None.*

PyPy

Фреймворк разрабатывался на языке *Python*. Это делает его код наглядным и легко расширяемым. Недостатком *Python* является низкая производительность, что не имеет веса до тех пор, пока пользователь не решит запустить объемный тест.

В таких случаях, можно воспользоваться интерпретатором *PyPy* [7] – альтернативой стандартному *CPython* интерпретатору. Поскольку он соответствует спецификации языка *Python*, *PyPy* не требует никаких изменений в коде и может предложить кратное улучшение в скорости благодаря своим особенностям.

Вот как устроен *PyPy*:

- Исходный код написан на языке *RPython*.
- Цепочка инструментов перевода *RPython* применяется к коду, что в основном делает код более эффективным. Он также компилирует код в машинный.
- Создается двоичный исполняемый файл.

PyPy использует *Jit*-компиляцию. Таким образом:

1. Определяются наиболее часто используемые компоненты кода, такие как функция в цикле.
2. Эти части преобразуются в машинный код во время выполнения.

3. Сгенерированный машинный код оптимизируется.
4. Начальная версия заменяется оптимизированной версией машинного кода.

Также в *PyPy* оптимизирована сборка мусора. В *CPython* используется подсчет ссылок. Количество ссылок увеличивается при каждом обращении к объекту и уменьшается при разыменовании объекта. Когда число ссылок большого дерева объектов становится равным нулю, все связанные объекты освобождаются. Дополнительно к этому устроен так называемый циклический сборщик мусора, благодаря которому исключаются случаи, с которыми не справляется подсчет ссылок. Он ходит по всем объектам в памяти, идентифицирует все доступные объекты и освобождает недостижимые объекты, так как они больше не эффективны.

PyPy не использует подсчет ссылок. Он периодически ходит по живым объектам. Это дает *PyPy* некоторое преимущество перед *CPython*, так как он не беспокоится о подсчете ссылок, что делает общее время, затрачиваемое на управление памятью, меньше, чем в *CPython*.

PyPy рационально использовать на крупных тестах и только на них. Он выполняет множество вещей, чтобы код работал быстрее. Если тест мал, то эти накладные расходы окажутся значимыми.

Таким образом если не предполагается запускать объемные тесты, то достаточно *CPython* интерпретатора. Иначе, рекомендуется использовать *PyPy* интерпретатор. Экспериментально оказалось, что он сокращает время работы программы в 3-4 раза.

3.4 Вывод данных

Помимо вывода в стандартный поток вывода был реализован вывод в *csv* файл и вывод в *dot* файл. Также использовалось журналирование.

CSV

CSV файл формируется из строк вида: `logic1.name logic2.name relation`. Строки составляются из всех элементов таблицы relations экземпляра класса *Transitivity*, взятой на момент окончания всех сравнений.

DOT

Для создания представления результата в виде графа был использован модуль *Graphviz* [8]. Получаемые орграфы соответствуют диаграммам Хассе для рассматриваемых соотношений.

Здесь фигурирует полученный в результате анализа экземпляр класса *Transitivity*. Сначала с помощью *find_functions* находятся все эквивалентные логики.

Записываются вершины графа:

- Минимальные логики располагаются снизу, самые сильные – сверху (т.е. используется флаг *rankdir = BT*).
- Эквивалентные логики располагаются в одном и том же узле.
- Отношения вложения, выполненные по транзитивности, не отображаются.
- Ребра исходят из вкладываемой логики во вкладывающую.

Журналирование

В файл *debug.log* выводится информация о ходе работы программы. Записываются случаи, не удалось вложить одну логику в другую. Есть такой ряд случаев:

- Если вложение $X \subseteq Y$ опроверглось тестом на неразличимые термы, тогда выводится:

  value(s) <перечисление> are discernible in X, indiscernible in Y on W

- Если провалился тест ОЗ, тогда записывается:

X returns <значение> not generated in Y

- Если тест на открытые множества сузил перебор по функциям, и оказалось, что из ограниченного набора данную функцию не построить, тогда записывается:

  function <имя> in X cannot be constructed in the basis [<функции>] of Y

- Если перебором не удалось построить какую-то из функций логики *X*, тогда записывается:

  (brute-force) function <имя> in X cannot be constructed in the basis of Y

Результаты удачных сравнений никак не комментируются.

4 Тестирование

В этом разделе описано, как контролировалось качество разрабатываемого кода, какие структуры программа проанализировала, и какие результаты получились.

4.1 Юнит-тесты

В ходе разработки, при добавлении новых возможностей программы, проводились тесты для того, чтобы избежать появления ошибок на уже пройденных этапах, или определить эффективность вновь добавленной оптимизации на старых задачах.

Для этого использовался модуль *unittest*. С его помощью постепенно разрабатывались юнит-тесты. *Unittest* был удобен своим интерфейсом: можно быстро запустить конкретные тесты, либо запустить сразу все; показывается, какие тесты прошли, какие не прошли, если тест не прошел, то понятно выводится подслучаи, где вышли ошибки. Также показывается время выполнения теста.

Рассмотрим созданный класс *TestAlgorithm(unittest.TestCase)*. В нем прописан метод *go*, использующийся в большинстве тестов для их запуска и сами тесты.

4.2 16 паралогик Левина-Микенберг

Первой контрольной точкой было достижение уже имеющегося результата: Структура вложения первых 16-ти логик Левина-Микенберг.

Для начала, в силу того, что несколько логик из этих 16 способны породить большое количество функций, алгоритм испытывался на нескольких из них. Был получен результат как на листинге А1 в приложении.

На этот момент еще не шла речь об оптимизациях и алгоритм напрямую перебирал всевозможные варианты композиции функций.

Далее была применена оптимизация *Своевременная остановка* и результат стал выдаваться значительно быстрее. Был произведен запуск алгоритма на всех 16-ти логиках. Нужный результат был достигнут. Он отображен на рисунке 12.

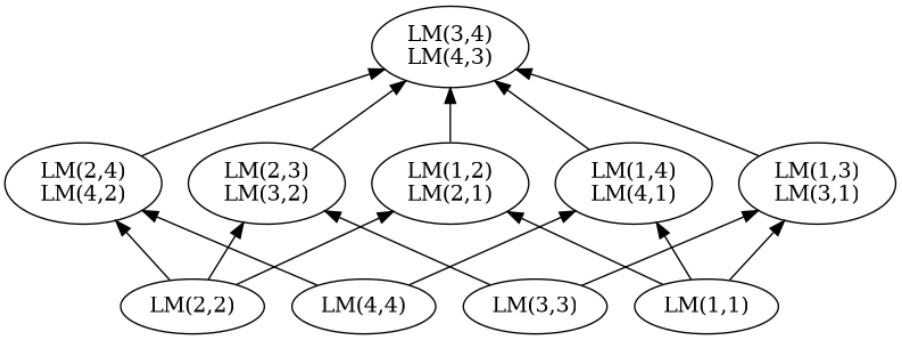

Рисунок 12. Результат на 16 паралогиках Левина-Микенберг

Хоть результат был правильным, но алгоритм получал его долго: всего нужно было произвести 120 попарных сравнений логик а количество возможных функций, а количество возможных функций в каждой из них 2^{16} (в силу того, что область значений этих функций ограничивается только двумя логическими значениями: 0 и 1). В логиках *LM(3,4)* и *LM(4,3)* могут порождаться все функции в {0, 1}. Тогда пришла очередь для следующих оптимизаций.

Была применена оптимизация Коммутативность. Она улучшила время работы, но не значительно (благодаря оптимизации была отсечена лишь малая часть избыточных порождаемых функций).

Также была разработана проверка на неразличимость значений. А вот эта оптимизация дала хороший прирост в производительности. Результат получался в разы быстрее. Как обнаружилось впоследствии, при воплощении уже всех

имеющихся оптимизаций, на этих 16-ти логиках только проверка на неразличимость значений отображалась в отчетах о работе программы. Часть отчета программы отображена на листинге А2 в приложении. На основе данного испытания в *TestAlgorithm* был создан *test_LM_1to4*.

4.3 64 паралогики Левина-Микенберг

Следующим этапом развития алгоритма было получение результата на всех 64 логиках Левина-Микенберг. Только с имеющимися оптимизациями алгоритм работал непозволительно долго. Нужно было произвести 2016 сравнений, к тому же область значений здесь шире (все 4 имеющихся значения), количество cnt возможных функций оценивается: $2^{16} < cnt < 2^{32}$.

Проблема возникает, например, при сравнении логик $LM(1,5)$ и $LM(1,6)$ с логиками $LM(3,4)$ и $LM(4,3)$. Поскольку все логические значения в этих логиках различимы (хотя в логиках $LM(1,5)$ и $LM(1,6)$ различимость обеспечивается исключительно за счет отрицания), фреймворк пытается решить проблему полным перебором (который обречен на неудачу и поэтому сильно затянется). Нужно подсказать алгоритму, что в таких случаях вложение, скажем, $LM(1,5)$ в $LM(3,4)$ проверять нет смысла. В логике $LM(3,4)$ все функции действуют в множество $\{0,1\}$, а в логике $LM(1,5)$ имеется функция (отрицание) с более широкой областью значений.

Здесь настало время разработать оптимизации *вложение области отображения*, *область значений 1* и *область значений 2*. Они позволяют отсечь проверку таких вложений.

С этими оптимизациями скорость работы значительно улучшилась. В журнале о работе программы появлялись сообщения о применении оптимизаций. Фрагмент отчета о работе программы приведен на листинге А3 в приложении.

Из 2016 пар пришлось сделать 108 полных переборов. При этом, *область значений 1* была эффективна 208 раз, а *область значений 2* – 464 раза, *вложение области отображения* – 736 раз.

Также в на этом этапе в *TestAlgorithm* были добавлены тесты *test_LM_15_16*, *test_LM_115* и *test_LM_17* для проверки в будущем того, что фреймворк на них не начнет работать дольше чем ожидалось.

Результат, полученный при анализе 64 паралогик Левина-Микенберг расположен на рисунке 13:

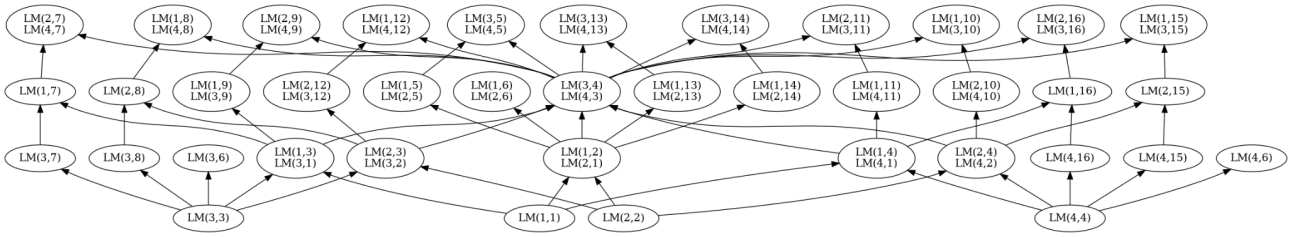

Рисунок 13. Результат на 64 паралогиках Левина-Микенберг

Отметим, что результаты по всем перечисленным тесты были проведены с учетом использования алгоритма *AsymmetricComparator.* При сравнении результатов полученного графа отношений с помощью *AggregatedComparator,* и на 16-ти логиках и на 64 логиках были получены те же результаты.

4.4 Тесты на супремумы

На основе предыдущего теста были проведены тесты, в которых проверяется отношение вложения между теми логиками, которые оказались включающими и логиками, объединяющими логики, которые были включены. Результаты расположены на листинге А4 в приложении.

Таким образом, для каждого вложения подтвердилось существования супремума как объединения пары логик, внутри того же самого множества 64 логик Левина-Микенберг.

4.5 Альтернативные паралогики Левина-Микенберг

Предполагалось, что есть альтернативное представление паралогик Левина-Микенберг с базисами без функций *And* и *Or*. С помощью разработанного фреймворка удалось также проверить эту гипотезу.

Утверждение: Всякая логика из класса 64 логик *LM(i,j)* с базисом $\rightarrow_i, \wedge_i, \vee_i, \neg_j$ эквивалентна логике *LM(i,j)alt* с базисом $\rightarrow_i, \neg_j$.

Доказательство: В каждый i-ый набор из наборов альтернативных логик добавляем логику *LM(i,i)* стандартную (с дизъюнкцией и конъюнкцией). Она самая слабая из всех логик вида *LM(i,k)*, о чем мы уже знаем из предыдущих тестов. Поэтому достаточно проверить, что для каждой из альтернативных *LM(i,k)alt* в нее вкладывается стандартная *LM(i,i)*. Это означает, что i-е дизъюнкция и конъюнкция выводимы во всех альтернативных базисах. Результаты проверок отображены на рисунках 14, 15, 16 и 17 :

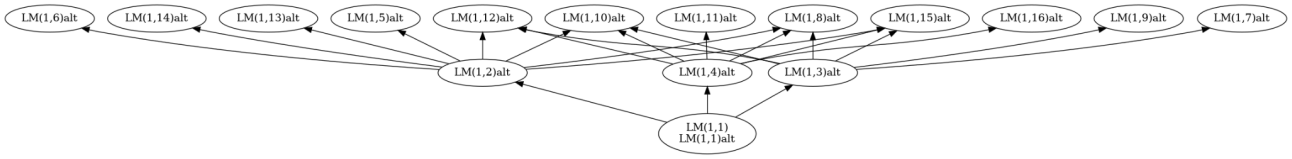

Рисунок 14. Первый набор логик

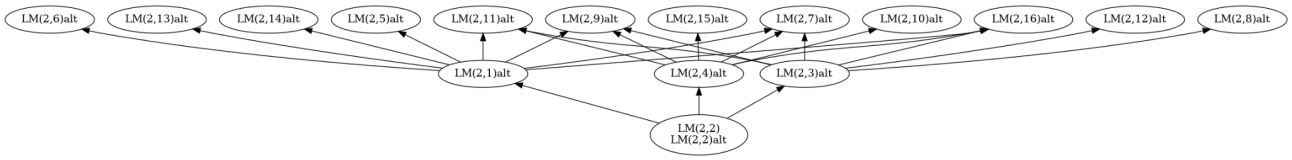

Рисунок 15. Второй набор логик

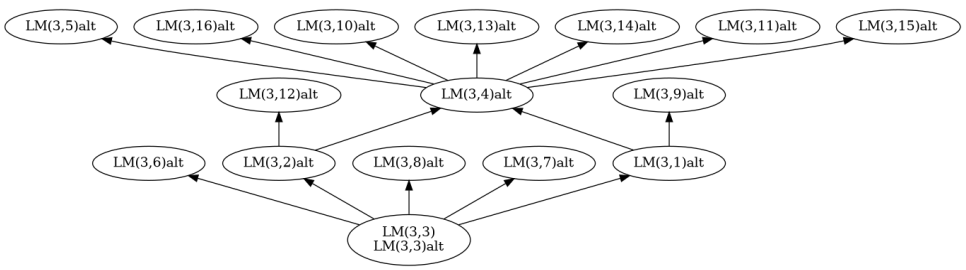

Рисунок 16. Третий набор логик

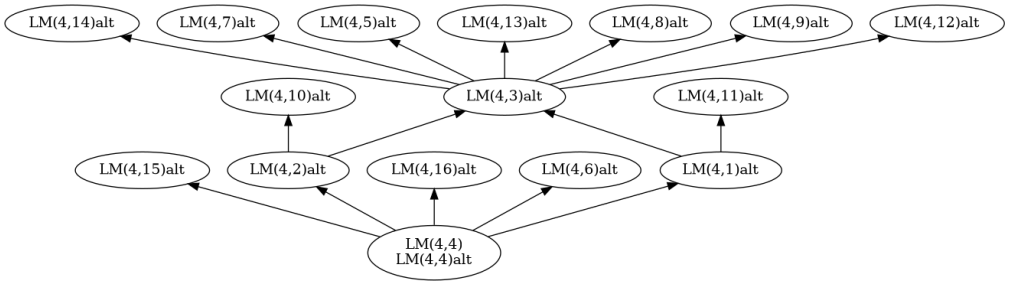

Рисунок 17. Четвертый набор логик

Таким образом, все нужные вложения выполнены, предположение верно. □

Теперь можно проверить, совпадают ли полные результаты анализа связей между альтернативными логиками и стандартными. Посмотрим на получившийся граф вложения альтернативных логик. Он на рисунке 18.

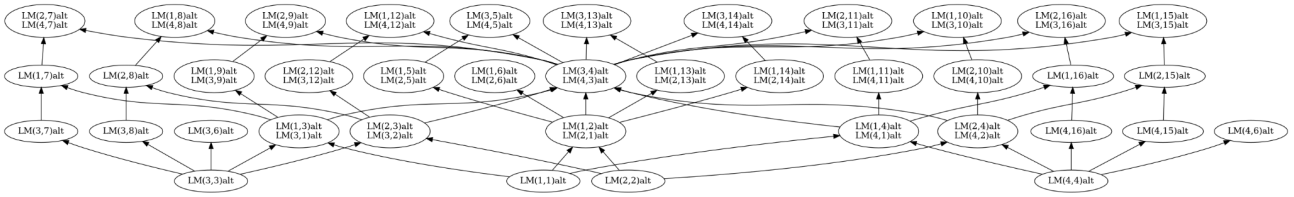

Рисунок 18. Результат на 64 альтернативных паралогиках

Замечается визуальная схожесть, но необходимо точное сравнение. Так были построены csv файлы, содержащие результаты анализа стандартного и альтернативного набора логик (предусмотрена сортировка строк при создании таких файлов), напомним, что записи в csv файлах имеют следующий вид:

```
<logic1_name> <logic2_name> <relation>
```

Третье значение было построчно сравнено. Результаты оказались эквивалентными.

На основе этой проверки в *TestAlgorithm* был разработан тест *test_compare_LM*.

5 Руководство пользователя

Для начала необходимо подготовить данные. Файлы с логиками должны быть собраны в каталог. Далее этот каталог необходимо разместить в директорию *data*.

Рассмотрим интерфейс взаимодействия с программой. Точкой запуска является файл *main.py*. Аргумент – имя каталога с входными файлами. Результат работы программы придет в стандартный поток вывода в виде последовательности строк: *<имя логики 1> and <имя логики 2> : отношение*.

Флаги:

- *-csv* – создание *csv* файла по результату.
- *-dot* – создание *dot* файла по результату. Все такие файлы будут сохраняться в в директорию *results*.
- *-aggr* – использование алгоритма перебора *AggregatedComparator* (по умолчанию используется алгоритм *AsymmetricComparator*).
- *-h* выводит подсказку как на листинге 5.

```
usage: main.py [-h] [-csv] [-dot] [-aggr] input

positional arguments:
  input       name of dir in DATA_PATH with mvlog files

optional arguments:
  -h, --help  show this help message and exit
  -csv        if you need to generate csv file
  -dot        if you need to generate dot file
  -aggr       if you want to run AggregatedComparator
```

Листинг 5. Подсказка

Сведения о ходе работы программы записываются в файл *debug.log*, который перезаписывается при каждом запуске. Формат сообщения в логе таков:

<Время> <Уровень сообщения>: <Сообщение>

Если вычисления предполагаются объемными, то рекомендуется использовать интерпретатор *PyPy3*, который позволит в разы сократить время работы кода.

ЗАКЛЮЧЕНИЕ

В ходе работы были выполнена поставленная задача – был разработан каркас, позволяющий сравнивать функциональные свойства конечнозначных логик. Реализованный продукт удобен тем, что пользователь по своему усмотрению может выбирать алгоритм сравнения логик, быстро разобраться в программном коде и и легко дополнить желаемый функционал.

Дополнительно был реализован вывод результатов в *csv* и *dot* файлы, что позволяет пользователю удобно анализировать итоги работы фреймворка.

Были проведены объемные тесты. Впервые был построен граф функциональной вложенности 64-х логик Левина-Микенберг. Была доказана гипотеза об избыточности конъюнкции и дизъюнкции в базисах этих логик.

Дальнейшая работа над фреймворком возможна в следующих направлениях. Во-первых, можно расширить список оптимизаций, эффективно отсекающих ветви перебора при проверке функциональной выразимости. Во-вторых, можно расширить журналирование, добавив вывод в лог информацию о способе построения логических операций в заданном базисе. В-третьих, можно добавить интерфейсы фреймворка с другими инструментами анализа.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Колмогоров А. Н., Драгалин А. Г.. Введение в математическую логику. Издательство Московского университета 1982 г.
2. Lewin R.A., Mikenberg I.F. Literal-paraconsistent and literal-paracomplete matrices. Math. Log. Quart. 2006. Vol. 52. No. 5. P. 478–493
3. Трехзначная логика Бочвара и литеральные паралогики. А. С. Карпенко, Н. Е. Томова. Российская академия наук. Институт философии. 2016
4. Г. Биркгоф. Теория решеток. Москва “Наука”. Главная редакция физико-математической литературы 1984.
5. Л. Ю. Девяткин. О подлинно паранепротиворечивых и подлинно параполных многозначных логиках. Логические исследования. 2019. Т. 25. No 2. С. 26–45
6. REFAL-5 programming guide & reference manual Valentin F. Turchin The City College of New York New England Publishing Co. Holyoke 1989
7. RealPython. PyPy: Faster Python With Minimal Effort. Jahongir Rahmonov. Режим доступа: realpython.com/pypy-faster-python/
8. Документация Graphviz. Режим доступа: graphviz.org/documentation/

ПРИЛОЖЕНИЕ А

```
LM(1,4) LM(1,2) non-comparable

LM(1,4) LM(1,1) embedded

LM(1,4) LM(1,3) non-comparable

LM(1,4) LM(1,1)alter embedded

LM(1,2) LM(1,1) embedded

LM(1,2) LM(1,3) non-comparable

LM(1,2) LM(1,1)alter embedded

LM(1,1) LM(1,3) embeds

LM(1,1) LM(1,1)alter equivalent

LM(1,3) LM(1,1)alter embedded
```

Листинг А1. Первый результат

```
values {('1', 'D')} are discernible in LM(2,1), indiscernible in LM(4,2)
values {('D', 'T')} are discernible in LM(4,2), indiscernible in LM(2,1)
values {('0', 'D'), ('1', 'T')} are discernible in LM(4,4), indiscernible
in LM(3,3)
values {('0', 'T'), ('1', 'D')} are discernible in LM(3,3), indiscernible
in LM(4,4)
values {('0', 'D')} are discernible in LM(4,2), indiscernible in LM(3,1)
values {('1', 'D')} are discernible in LM(1,4), indiscernible in LM(4,2)
values {('0', 'T')} are discernible in LM(4,2), indiscernible in LM(1,4)
values {('1', 'D')} are discernible in LM(1,3), indiscernible in LM(4,2)
```

Листинг А2. Отчет о работе программы на 16-ти логика

```
function Not in LM(2,13) cannot be constructed in the basis ['Not'] of
LM(2,6)
values {('D', 'T')} are discernible in LM(2,6), indiscernible in LM(2,13)
LM(2,6) returns {'T'} not generated in LM(4,14)
(brute-force) functions {
          Not
          ('1', '0', '1', '0')
     } in LM(2,3) cannot be constructed in the basis of LM(2,6)
LM(2,6) returns {'T'} not generated in LM(4,7)
```

Листинг А3. Отчет о работе программы на 64-ти логиках

```
LM(3,5) LM(1,5)+LM(3,4) : equivalent
LM(2,7) LM(1,7)+LM(3,4) : equivalent
LM(2,9) LM(1,9)+LM(3,4) : equivalent
LM(2,11) LM(1,11)+LM(3,4) : equivalent
LM(1,16) LM(4,16)+LM(1,4) : equivalent
LM(2,8) LM(3,8)+LM(2,3) : equivalent
LM(2,15) LM(4,15)+LM(2,4) : equivalent
LM(1,7) LM(3,7)+LM(1,3) : equivalent
LM(3,13) LM(1,13)+LM(3,4) : equivalent
LM(1,12) LM(2,12)+LM(3,4) : equivalent
LM(1,8) LM(2,8)+LM(3,4) : equivalent
LM(3,14) LM(1,14)+LM(3,4) : equivalent
LM(1,15) LM(2,15)+LM(3,4) : equivalent
LM(2,16) LM(1,16)+LM(3,4) : equivalent
LM(1,10) LM(2,10)+LM(3,4) : equivalent
```

Листинг 4. Тесты на супремумы