СОДЕРЖАНИЕ

Номер 5, 2020

ПРОГРАММНАЯ ИНЖЕНЕРИЯ, ТЕСТИРОВАНИЕ И ВЕРИФИКАЦИЯ ПРОГРАММ

ЯЗЫКИ, КОМПИЛЯТОРЫ И СИСТЕМЫ ПРОГРАММИРОВАНИЯ

БАЗЫ ДАННЫХ, ХРАНИЛИЩА ДАННЫХ

ПОПРАВКА

CONTENTS

No. 5, 2020

SOFTWARE ENGINEERING, TESTING AND VERIFICATION

PROGRAMMING LANGUAGES, COMPILERS, AND INTEGRATED DEVELOPMENT ENVIRONMENT

DATABASES, DATA WAREHOUSES

CORRECTION

ПРОГРАММИРОВАНИЕ, 2020, № 5, с. 3–10

ПРОГРАММНАЯ ИНЖЕНЕРИЯ, ТЕСТИРОВАНИЕ И ВЕРИФИКАЦИЯ ПРОГРАММ

УДК 004.42

НЕКОТОРЫЕ АСПЕКТЫ ПОСТРОЕНИЯ АССОЦИАТИВНОЙ ПАМЯТИ НА НЕЙРОННОЙ СЕТИ ХОПФИЛДА

© 2020 г. Ю. Л. Карпов[a,*], Л. Е. Карпов[b,c,], Ю. Г. Сметанин[d,e,***]**

[a] *ООО Люксофт Профешнл*
123060 Москва, 1-й Волоколамский проезд, 10, Россия

[b] *Институт системного программирования им. В.П. Иванникова РАН*
109004 Москва, ул. Солженицына, 25, Россия

[c] *Московский Государственный университет имени М.В. Ломоносова*
119991 Москва, Ленинские горы, 1, Россия

[d] *Федеральный исследовательский центр "Информатика и управление" РАН*
119333 Москва, ул. Вавилова, д. 44, кор. 2, Россия

[e] *Московский Физико-Технический институт 141701 Московская область,*
г. Долгопрудный, Институтский пер., 9, Россия

*****E-mail: y.l.karpov@yandex.ru*

******E-mail: mak@ispras.ru*

*******E-mail: ysmetanin@rambler.ru*

Поступила в редакцию 05.03.2020 г.
После доработки 25.03.2020 г.
Принята к публикации 11.04.2020 г.

Уже после сдачи в печать этой статьи безвременно ушел из жизни вдохновитель нашей работы Юрий Геннадьевич Сметанин. Он не оставлял своей работы до самого конца, уже по телефону из больницы обсуждая планы следующих исследований и экспериментов. Светлая память о нашем товарище, настоящем ученом и очень хорошем человеке останется с нами навсегда.

Рассматривается реализация ассоциативной памяти, в основе которой находится нейронная сеть Хопфилда. В предлагаемом подходе адреса в памяти трактуются как обучающие векторы нейронной сети. Эффективность поиска информации в памяти напрямую связана с решением проблемы переобучения. Предлагается метод разделения обучающих и входных векторов сети на части, обработка которых требует меньшего числа нейронов. Описываются результаты серии экспериментов, проводившихся на моделях сетей Хопфилда с различным числом нейронов, обучавшихся с разным количеством векторов и работавших в условиях различного уровня шума.

DOI: 10.31857/S0132347420050027

1. ВВЕДЕНИЕ

Программные модели нейронных сетей представляют собой объекты, исключительно интересные для исследования. С одной стороны, эти модели суть системы программного обеспечения, проектирование, разработка и эксплуатация которых подчиняется всем законам, правилам и стандартам, которым подчиняются другие программные системы. С другой стороны, особенностью этих моделей является то, что в них, как, пожалуй, ни в каких других системах не достигают такой степени важности вопросы структурной организации многочисленных составляющих их компонентов. При том, что каждый отдельный компонент нейронной сети – это достаточно сложно организованный вычислительный комплекс, большое число таких компонентов в сети и разнообразие методов организации их взаимодействия делают проблемы выбора и последующей проверки его правильности чрезвычайно важными и трудными.

Авторы уже привлекали внимание к проблеме организации тестирования структуры нейронных сетей (см. [1–3]).

Мы рассматриваем конкретную модель рекуррентной нейронной сети – модель Хопфилда [см. 4], то есть сеть из n нейронов с полным набором симметричных связей, в которой действуют правила

обновления нейронов – $x_i(t+1) = \mathrm{sgn}\left(\sum_{j=1}^{n} w_{ij} x_j(t)\right)$,

где n – число нейронов, w_{ij} – вес связи между нейронами i и j, $w_{ji} = w_{ij}$. Доказано, что при асинхронном обновлении нейронов эта сеть всегда сходится к состоянию равновесия, в котором вектор значений нейронов перестает изменяться. Число состояний равновесия называется емкостью сети. При любом начальном состоянии сети она на некотором шаге попадает в одно из состояний равновесия. Для каждого состояния равновесия областью ее притяжения называется множество состояний сети, из которых она в ходе обновления попадает в это состояние. Таким образом, можно считать, что область притяжения – это набор входных векторов с такими искажениями, которые данная нейронная сеть исправляет. Именно поэтому нейронную сеть Хопфилда часто называют ассоциативной памятью. Такая сеть разбивает множество допустимых входных векторов на классы, каждый из которых соответствует своему эталону – состоянию равновесия сети, и распознает принадлежность входного вектора классу.

2. ЕМКОСТЬ, ТОЧНОСТЬ РЕШЕНИЯ, СОСТОЯНИЯ РАВНОВЕСИЯ НЕЙРОННОЙ СЕТИ

Для построения количественных характеристик способности конкретной модели нейронной сети к распознаванию естественно использовать число классов, на которые эта сеть может эффективно разделить множество допустимых векторов. Эта величина называется емкостью данной модели сетей. Емкость является вероятностной характеристикой, поскольку зависит от вида классов, то есть конкретных параметров нейронной сети. Как правило, используются асимптотические значения емкости при $n \to \infty$. Определения емкости, соответствующие различным моделям и различным применениям, можно найти в работе [5].

Пусть дан эталонный набор из p векторов $x^1, \ldots, x^p$ и требуется построить матрицу весов синапсов, при которых каждый из этих векторов будет состоянием равновесия сети Хопфилда. Точное решение этой задачи получается при использовании псевдообратной матрицы для построения матрицы весов,

$$W = X(X^T X)^{-1} X^T$$

где X – матрица размера $n \times p$, столбцами которой являются векторы из данного набора.

Очевидно, при такой матрице все эти векторы X будут состояниями равновесия. Для n векторов в общем положении матрица W является невырожденной, откуда следует, что емкость модели нейронной сети Хопфилда – не менее n.

К сожалению, платой за точность решения с использованием обращений матриц является сложность вычислений обратной матрицы. Поэтому на практике обычно используют более простое правило обучения, позволяющее обойтись без сложных вычислений, то есть полностью использовать достоинства нейронных сетей. Здесь будет рассмотрен распространенный метод обучения с учителем для построения однослойных рекуррентных нейронных сетей Хопфилда – правила Хебба (см. [6]).

Правила Хебба были впервые предложены для объяснения динамики биологических нейронных сетей и выражают принцип “если два нейрона часто загораются и гаснут вместе, связь между ними надо усилить”. Из этого принципа для искусственных нейронных сетей получено простое правило обучения – метод внешнего произведения, который работает следующим образом (для простоты рассмотрим двоичный случай, значения нейронов – +1 или –1).

Пусть задан набор векторов $a^1 = (a_i^1), \ldots, a^p = (a_i^p)$ и требуется построить матрицу весов синапсов W, для которой эти векторы являются положениями равновесия, $\mathrm{sgn}(Wa^k) = a^k$. Для каждого вектора $a^1, \ldots, a^p$ строится матрица W^k с элементами $w_{ij}^k = a_i^k a_j^k$. $w_{ij}^k = \begin{Bmatrix} a_i^k a_j^k, i \neq j \\ 0, i = j \end{Bmatrix}$.

Очевидно, $\mathrm{sgn}(W^k a^k) = \mathrm{sgn}\left(\sum_{j=1}^{n} a_i^k a_j^k a_j^k\right) = \mathrm{sgn} a_i^k$,

то есть a является состоянием равновесия для сети с матрицей W^k. С другой стороны, если вектор b ортогонален a^k, то $(W^k b) = 0$. Следовательно, для ортогональных наборов векторов a^k матрица $W = \sum_{k=1}^{p} W^k$ обладает требуемым свойством. В общем случае гарантировать этого нельзя, и оценки корректности метода внешнего произведения являются вероятностными.

Статистическая механика модели Хопфилда, впервые проанализированная в [7], основанная на использовании аналогий с моделью спиновых стекол, позволила получить достаточно интересные результаты о работе модели как ассоциативной памяти. Основная идея заключается в анализе величины $M^\mu = \frac{1}{n}\sum_{i=1}^{n} s_i^\mu s_i$, характеризующей сходство произвольного текущего состояния с μ-ым эталоном. Очевидно, что для состояний, не коррелирующих с эталонами, эта величина имеет порядок

$1/\sqrt{N}$, а для состояний, отличающихся от данного эталона в небольшом числе координат, она близка к единице. Число обучающих векторов, на которых можно обучить модель Хопфилда с помощью метода внешнего произведения таким образом, чтобы все они стали состояниями равновесия, в предположении, что обучающие векторы выбраны по схеме Бернулли, – $O(n/\log_2 n)$ (см. [8, 9]). В этом случае, как следует из сказанного раньше, возникают посторонние состояния равновесия.

Вид областей притяжения в сети может быть сложным, эти области даже могут быть несвязными (см. [10]). Вероятностный характер обучения приводит к тому, что области притяжения посторонних состояний равновесия могут искажать области притяжения истинных состояний равновесия. Для ассоциативной сети желательно, чтобы истинные состояния равновесия, соответствующие обучающими эталонам, имели по возможности большие области притяжения, а посторонние были как можно меньше, а в идеале вообще отсутствовали.

Отсюда следует, что эффективность нейронной сети становится малой не только в случае, когда число эталонов слишком велико, и неизбежно такое сходство между ними, которое приводит к искажениям вместо коррекции входных векторов, но и в случае, когда эталонов слишком мало. В последнем случае неизбежно появление больших посторонних областей притяжения, из-за чего трудно оценивать правильность полученных результатов, а кроме того, сохранить существенные различия между обучающими эталонами можно и при их преобразовании в представления меньшей размерности, а, следовательно, появляется возможность снизить размер (сложность) нейронной сети, требуемой для решения поставленной задачи.

Авторами исследовался случай, когда число эталонов значительно меньше критического, и для построения ассоциативной памяти, запоминающей эти эталоны и корректирующей их искажения, оказывалось достаточно значительно меньшей размерности сети. В этом случае естественно использовать метод кодирования со сжатием, однако его реализация требует решения задачи о выборе кода с оптимальным сочетанием размерности кодовых слов и способности к исправлению ошибок. Кроме того, необходимо обеспечить выполнение следующего не очень простого для проверки условия: малые искажения, исправляемые сетью Хопфилда с исходными векторами, останутся исправляемыми и сетью Хопфилда, работающей с закодированными представлениями.

Авторами предложен более простой подход к уменьшению сложности нейронной сети в этом случае, основанный на разбиении каждого эталона на r частей одинаковой длины с приписыванием каждой части дополнительных вспомогательных компонентов, указывающих соответствующий эталон, позицию части в этом эталоне и, возможно, компонента для выравнивания длин. Обучение сети производится с помощью правил внешнего произведения на новом наборе эталонов, полученным в ходе разбиения исходного набора. При этом вспомогательные компоненты должны точно приписываться и поступающим на вход обученной сети анализируемым векторам, также разбитым на части; эти компоненты не могут изменяться в ходе нейросетевой обработки. По окончании работы нейронной сети исправленные ею части сливаются в единый исправленный вектор.

При этом новое число эталонов $p' = rp$, новая длина векторов $n' = n/r$, а значение r подбирается таким образом, чтобы новое значение $\alpha' = pr^2/n$ было меньше критического.

3. ЭКСПЕРИМЕНТЫ С НЕЙРОННЫМИ СЕТЯМИ

Тестирование немыслимо без эксперимента. Тестирование структуры нейронной сети такой же эмпирический процесс, как и тестирование традиционного программного обеспечения. Несмотря на наличие большого числа теоретических разработок, повышающих отдачу от проведения этого необходимого для создания эффективных, надежных, удобных программ и программных моделей (см. [11–14] и многие другие работы, посвященные тестированию), эксперимент в тестировании не теряет своей важности.

Исследуя возможности влияния на структуру нейронной сети, авторы провели несколько серий экспериментов, позволивших уточнить представления о поведении сетей в различных условиях и сделать важные выводы.

В качестве объекта исследований были выбраны сети Хопфилда, на входы которым подавались одноразрядные значения, трактуемые как числа –1 и +1. Количество входов задавалось разным в разных экспериментах и менялось от 13 до 960.

В рамках проводимого исследования нейронная сеть трактовалась нами, как модель ассоциативной памяти, соответственно, все эксперименты планировались нами как попытки запомнить некоторое количество чисел, а затем выполнить серию чтений из памяти при получении “правильных” запросов. “Правильность” запроса определялась как наличие по соответствующему адресу в ассоциативной памяти хранящейся информации. Эксперименты намеренно осложнялись тем, что в качестве запрашиваемых адресов на вход подавались не только те “ассоциативные” адреса, по которым первоначально осуществлялась запись ин-

формации, но и другие, отличающиеся от “правильных” в той или иной степени.

При проведении экспериментов нас особенно интересовала возможность снижения “ширины” сети без потери качества запоминания (или с управляемым снижением этого качества) в случае, когда число запоминаемых объектов невелико по сравнению с априорными оценками емкости сети. Очевидно, что снижение числа нейронов в сети должно приводить к повышению эффективности программных моделей, однако, с аппаратно реализованными нейронными сетями такой простой зависимости не существует. Значительная часть затрат на аппаратную реализацию нейронной сети осуществляется еще на стадии ее проектирования и физического построения. В дальнейшем затраты в основном связаны с энергопотреблением во время работы сети. Очевидно также, что снижение числа нейронов, которого удается добиться при выборе характеристик сети, необходимых для решения поставленной задачи, выгодно и для аппаратной реализации сетей, и для построения их программных моделей.

Поскольку программная работа с нейронными сетями больших размерностей связана с большими затратами времени, в наших экспериментах с сетями с различным количеством нейронов векторы, подававшиеся на вход сетям, различались не только разрядностью (что очевидно), но и критериями, по которым они генерировались.

Сетям небольшой размерности (в наших экспериментах – это сети из 13 и 23 нейронов) подавались на вход полные наборы всех возможных векторов соответствующих размерностей – 8192 разряда в первом случае и 8388608 во втором. В этих наборах случайным образом авторы выбирали необходимое для проведения эксперимента количество векторов, объявлявшихся эталонными. Именно эти вектора запоминались в ассоциативной памяти, то есть использовались для обучения сетей.

Для более крупных сетей такой подход ввиду невозможности за разумное время обработать большое число входных векторов оказался неприемлемым, поэтому для них был предложен специальный механизм отбора эталонных векторов.

На сайте [15] была выбрана случайная последовательность, составлявшаяся при замерах метеорологических данных в Пекинском международном аэропорту. Некоторые из собранных данных (направление ветра) описывались в этом наборе как азимуты в виде кодов, соответствующих разным географическим направлениям с интервалом в 22,5 градуса, например, направление на север – это *N*, направление на северо-северо-запад – это *NNW*, и так далее. Фактически таким образом кодировались 16 различных направлений, которые мы стали трактовать как шестнадцать цифр в шестнадцатеричной системе счисления ($N \rightarrow$ ‘0’, $NNE \rightarrow$ ‘1’, ..., $NNW \rightarrow$ ‘f’). Тем самым, в нашем распоряжении оказалась последовательность из 35064 случайных шестнадцатеричных цифр. Их этих случайных последовательностей были составлены случайные векторы с размерами, соответствовавшими числу нейронов в исследовавшихся нейронных сетях, то есть в 32, 40, 256 и 960 разрядов.

Понятно, что генерация всех возможных входных векторов с соответствующим числом разрядов, хотя и несложна, но очень затратна по времени, поэтому на основе отобранных эталонов мы сгенерировали набор входных векторов, куда включили только те векторы, которые отличались от эталонов не более, чем в одной шестнадцатеричной цифре. При этом для некоторого приближения к более представительной выборке отличия в отдельных шестнадцатеричных цифрах допускались такие, чтобы можно было имитировать одно-, двух-, трех и четырехразрядный шум.

В экспериментах фиксировались следующие параметры сетей:

- длина входного вектора;
- число заданных обучающих (запоминаемых) векторов и верхняя оценка оптимального числа таких векторов;
- уровень шума (максимальное число разрядов, отличающих входные векторы от эталонов);
- число частей, на которые делились входные векторы при построении укороченных сетей;
- число входных векторов, поступивших на вход нейронной сети;
- доля верных решений (число входных векторов, которые были успешно преобразованы к одному из обучающих векторов);
- среднее число шагов, которые были выполнены в сети при обработке одного входного вектора;
- суммарное число шагов, выполненных сетью при обработке всех входных векторов, участвовавших в данном эксперименте.

Во время экспериментов проводились замеры размеров параметров сокращенных входных векторов (размеров информационной и служебных фрагментов векторов), определялась максимальная емкость ассоциативной памяти выбранной размерности, измерялось количество обработанных входных векторов, среднее число шагов, выполненных нейронных сетью для выяснения правильности или ложности полученного решения, либо фиксации нестабильности достигнутого состояния, а также общее число шагов, сделанных при обработке сетью всех входных векторов.

Каждый эксперимент проводился в двух вариантах. Сначала выполнялось моделирование рабо-

Таблица 1. Обучающие векторы в экспериментах с 13-разрядными нейронными сетями

Номер разряда		0	1	2	3	4	5	6	7	8	9	10	11	12
Обучающий вектор 1		−1	1	1	−1	−1	−1	1	1	1	−1	1	1	−1
Обучающий вектор 2		−1	−1	1	1	1	1	−1	1	1	−1	1	1	−1
Части вектора 1	0	−1	−1	−1	1	1	−1	−1	−1	1				
	1	−1	1	−1	−1	1	−1	−1	1	−1				
Части вектора 2	0	1	−1	−1	−1	1	1	1	1	−1				
	1	1	1	1	1	−1	1	1	−1	−1				

Таблица 2. Обучающие векторы в экспериментах с 23-разрядными нейронными сетями

Номера двоичных разрядов		0–2		3–6	7–10	11–14	15–18	19–22
Номер 16-ричного разряда		0		2	3	4	5	6
Обучающий вектор 1		4		4	5	c	b	8
Обучающий вектор 2		2		d	2	7	c	2
Номера двоичных разрядов		0	1	2–5	6–9	10–13		
Части обучающего вектора 1	0	−1	−1	4	4	5		
	1	−1	1	c	b	8		
Части обучающего вектора 2	0	1	−1	2	d	2		
	1	1	1	7	c	2		

ты полной нейронной сети с выбранным числом нейронов. Затем с теми же обучающими и входными векторами проводилась операция деления на части, и моделировалась сокращенная нейронная сеть с уменьшенным числом нейронов. Число частей выбиралось равным двум, лишь для сети из 960 нейронов оно составляло 4.

Каждая часть исходного обучающего вектора строилась из таких полей:

• поле номера обучающего вектора (двоичный логарифм от числа обучающих векторов p, с округлением вверх);

• поле номера части вектора (двоичный логарифм от числа частей r, на которые дробится вектор, с округлением вверх);

• информационное поле (число нейронов n, деленное на число частей, с округлением вверх).

Аналогично строились части входных векторов, при этом поле номера обучающего вектора заполнялось числом, соответствующим номеру обучающего вектора, отличавшегося от входного вектора в наименьшем числе позиций.

В таблице 1 показаны значения элементов обучающих векторов, подававшихся на вход нейронной сети, состоявшей из 13 нейронов. Результаты экспериментов с этой сетью собраны в таблице 2.

В таблице 3 показаны значения элементов обучающих векторов, подававшихся на вход нейронной сети, состоявшей из 23 нейронов. Эти значения представлены в виде 23-разрядных шестнадцатеричных чисел. Нулевое значение какого-либо двоичного разряда в этих числах соответствует элементу вектора, равного −1. Единичное значение двоичного разряда числа соответствует единичному элементу вектора. Результаты экспериментов с этой сетью собраны в таблице 2.

Уже по таким небольшим экспериментам с весьма небольшими нейронными сетями стали видны некоторые тенденции. И главное, что обратило на себя внимание, это сильная зависимость укороченных сетей от шума в поступающих на вход векторах. Даже относительно небольшие искажения во входной информации приводят к резким снижениям доли правильно распознанных векторов. Чтобы выявить другие проблемы и условия их возникновения, потребовалось провести несколько дополнительных экспериментов с сетями существенно большего размера. Результаты этих экспериментов показаны в таблицах 4 и 5.

Таблица 3. Результаты эксперимента с малоразмерными нейронными сетями

Длина входного вектора	13				23			
Число эталонов	2				2			
Шум (разрядов)	0		4		0		4	
Число частей		2		2		2		2
Информационная часть		7		7		12		12
Размер области номера эталона		1		1		1		1
Размер области номера части		1		1		1		1
Размер обрабатываемого вектора	13	9	13	9	23	14	23	14
Максимальная емкость памяти	4	3	4	3	6	4	6	4
Число обучающих векторов	2	4	2	4	2	4	2	4
Число входных векторов	8192				8388608			
Число обработанных векторов	2		1360		2		17710	
Доля верных решений (процентов)	100	100	78.23	5	100	50	99.21	19.09
Среднее число шагов сети	1	1	2.765	3.028	1	2	2	2.678
Полное число шагов сети	2	4	3760	8236	2	8	35420	94867

4. ВЫВОДЫ

Проведенные эксперименты позволили сделать некоторые выводы. У укороченных сетей более высокие требования к подбору обучающих векторов, при предлагаемом разделении могут возникать векторы с большей корреляцией, то есть увеличивается риск ложных ассоциаций.

Укороченные сети весьма чувствительны к количеству обучающих векторов, точнее при выборе обучающих векторов очень важно представлять себе, что их количество пропорционально возрастает с увеличением количества частей, на которые должны делиться обучающие и входные векторы. Пренебрежение этим может привести к проявлению проблем переобучения нейронной сети.

Укороченные сети более чувствительны к шумам, то есть при росте помех при передаче данных на вход нейронной сети может ухудшаться способность ассоциативной памяти исправлять некоторые погрешности, которые могут возникать при передаче информации в сеть на обработку.

Таблица 4. Результаты эксперимента с нейронными сетями

Длина входного вектора	32				40			
Число эталонов	3				3			
Шум (разрядов)	0		4		0		4	
Число частей		2		2		2		2
Информационная часть		16		7		20		20
Размер области номера эталона		2		1		2		2
Размер области номера части		1		1		1		1
Размер обрабатываемого вектора	32	19	32	9	40	23	40	23
Максимальная емкость памяти	7	5	7	5	8	6	8	6
Число обучающих векторов	3	6	3	6	3	6	3	6
Число обработанных векторов	841		10		811		12	
Доля верных решений (процентов)	100	0	100	0	100	100	100	41.67
Среднее число шагов сети	1	2.5	2	3.1	1	1	2	2.333
Полное число шагов сети	841	4205	20	62	811	1662	24	56
Время работы сети (тактов)	180	180	173	173	210	211	209	206

Таблица 5. Результаты эксперимента с нейронными сетями

Длина входного вектора	256				960			
Число эталонов	6				20			
Шум (разрядов)	0		4		0		4	
Число частей		2		2		4		4
Информационная часть		128		128		240		240
Размер области номера эталона		3		3		5		5
Размер области номера части		1		1		2		2
Размер обрабатываемого вектора	256	132	256	132	960	247	960	247
Максимальная емкость памяти	32	19	32	19	97	32	97	32
Число обучающих векторов	6	12	6	12	20	80	20	80
Число обработанных векторов	116		367		81020		4800	
Доля верных решений (процентов)	100	100	100	89.65	100	0	100	0
Среднее число шагов сети	1	1	2	1.631	1.4	13.875	2.27	13.89
Полное число шагов сети	116	232	367	1197	113428	4496610	10880	266751

В аппаратных реализациях нейронных сетей укорачиванием сети достигается экономия оборудования (но вряд ли экономится время решения задачи). В программных реализациях нейронных сетей укорачиванием сети достигается экономия времени решения задачи (хотя чуть меньшая, чем прямая пропорция с количеством частей из-за включения в укороченные входные векторы двух служебных полей). Хотя число шагов сети при уменьшении числа нейронов в ней возрастает, но каждый такой шаг выполняется с вовлечением в процесс гораздо меньшего числа нейронов. При моделировании операций умножения матрицы на вектор число выполняемых операций при этом уменьшается пропорционально второй степени числа частей, а это приводит к почти линейному сокращению времени моделирования.

В дальнейшем целесообразно провести исследование других способов входных представлений, наилучшим образом сочетающих требования к сложности сети и возможности ассоциативной памяти.

5. БЛАГОДАРНОСТИ

Авторы выражают благодарность Российскому фонду фундаментальных исследований, который поддержал эту работу своими грантами (проекты 18-07-00697-а, 18-07-01211-а, 19-07-00321-а и 19-07-00493-а).

СПИСОК ЛИТЕРАТУРЫ

1. *Карпов Ю.Л., Карпов Л.Е., Сметанин Ю.Г.* Адаптация общих концепций тестирования программного обеспечения к нейронным сетям. Программирование. 2018. т. 44. № 5. С. 43–56. DOI: / Yu.L. Karpov, L.E. Karpov, Yu.G. Smetanin. Adaptation of General Concepts of Software Testing to Neural Networks. Programming and Computer Software. 2018. V. 44. № 5. P. 324–334. DOI: https://doi.org/10.1134/S0361768818050031. https://doi.org/10.31857/S013234740001214-0
2. *Карпов Ю.Л., Карпов Л.Е., Сметанин Ю.Г.* Устранение отрицательных циклов в некоторых структурах нейронных сетей с целью достижения стационарных решений. Программирование. 2019. Т. 45. № 5. С. 25–35. DOI: / Yu.L. Karpov, L.E. Karpov, Yu.G. Smetanin. Elimination of Negative Circuits in Certain Neural Network Structures to Achieve Stable Solutions. Programming and Computer Software. 2019. V. 45. № 5. P. 241–250. DOI: https://doi.org/10.1134/S0361768819050025. https://doi.org/10.1134/S0132347419050029
3. *Карпов Ю.Л., Волкова И.А., Вылиток А.А., Карпов Л.Е., Сметанин Ю.Г.* Проектирование интерфейсов классов графовой модели нейронной сети. Труды ИСП РАН. 2019. Т. 31. Вып. 4. С. 97–112. / Karpov Yu.L., Volkova I.A., Vylitok A.A., Karpov L.E., Smetanin Yu.G. Designing classes' interfaces for neuron network graph model. Programming and Computer Software. 2020. V. 46, to be published. https://doi.org/10.15514/ISPRAS-2019-31(4)-6
4. *Hopfield J.J.* Neural networks and physical systems with emergent collective computational properties. Proc. Nat. Acad. Sci. USA. 1982. V. 79. P. 2554–2558.
5. *Smetanin Yu.G.* Neural networks as systems for recognizing patterns. In: Journal of Mathematical Sciences. 1998. V. 89. № 4. P. 1406–1457.
6. *Hebb D.O.* The Organization of Behavior. Wiley, 1948.
7. *Amit D.J., Gotfreund H., Sompolinski H.* Spin-Glass Models of Neural Networks, Phys. Rev. A. 1985. V. 32. P. 1007.
8. *McEliece R.J., Posner E.G., Rodemich E.R., Venkatesh S.S.* The Capacity of the Hopfield Associative Memory, IEEE Trans. Inf. Theory. 1987. V. 33. № 4. P. 461–482.
9. *Venkatesh S.S.* Computation and Learning in the Context of Neural Network Capacity, Neural Networks for Perceptions, Wechsler, H., Ed., Academic; Harcourt Brace Jovanovich. 1992. V. 2. P. 173–207.

10. *Makhoul J., El-Jaroudi A., Schwartz R.* Formation of Disconnected Regions with a Single Hidden Layer, Int. Joint Conf. on Neural Networks, Washington, DC. 1989. V. 1. P. 455–460.

11. *Майерс Г.* Искусство тестирования программ, М., Финансы и статистика, 1982 / Myers, G. J., The Art of Software Testing, John Wiley & Sons, 1979.

12. *Канер С., Фолк Д., Енг Кек Нгуен*. Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений, Киев, Диасофт, 2001 / Cem Kaner, Jack Falk, Hung Quoc Nguyen, Testing Computer Software, Second Edition, International Thompson Publishing Press, 1999.

13. *Тампе Л.* Введение в тестирование программного обеспечения, М.: Издательский дом “Вильямс”, 2003 / Louise Tamres, Introducing Software Testing, Addison Wesley, 2002.

14. *Бейзер Б.* Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем, Питер, 2004 / B. Beizer, “Software Testing Techniques,” Second Edition, Van Nostrand Reinhold Company Limited, 1990.

15. https://archive.ics.uci.edu/ml/datasets/Beijing+PM2.5+Data#

ПРОГРАММИРОВАНИЕ, 2020, № 5, с. 11–15

ПРОГРАММНАЯ ИНЖЕНЕРИЯ, ТЕСТИРОВАНИЕ И ВЕРИФИКАЦИЯ ПРОГРАММ

УДК 004.42

СХЕМЫ ОРГАНИЗАЦИИ “ЖИВОЙ” МИГРАЦИИ В ЦЕНТРАХ ОБРАБОТКИ ДАННЫХ

© 2020 г. В. А. Костенко[a],*, А. А. Чупахин[a],**

[a] *Московский государственный университет имени М.В. Ломоносова*
119991 Москва, Ленинские горы, 1, Россия

**E-mail: kost@cs.msu.su*

***E-mail:andrewchup@lvk.cs.msu.ru*

Поступила в редакцию 29.08.2019 г.
После доработки 01.11.2019 г.
Принята к публикации 01.11.2019 г.

В статье описаны схемы организации “живой” миграции в современных гипервизорах и проведен сравнительный анализ потенциальных возможностей схем по критериям: промежуток времени, после которого сервер-источник освободится; общее время миграции; промежуток времени, в течение которого в процессе миграции виртуальная машина недоступна; общее количество переданных данных в процессе миграции и уменьшение производительности мигрирующей виртуальной машины.

DOI: 10.31857/S0132347420050039

1. ВВЕДЕНИЕ

В процессе работы центров обработки данных (ЦОД) создаются новые или удаляются старые виртуальные машины (ВМ). При попытке разместить новую ВМ в ЦОД, может возникнуть следующая ситуация. Суммарного количества свободных физических ресурсов по всем запрошенным требованиям в соглашении о качестве обслуживания (SLA) хватает для размещения ВМ, но не существует физического ресурса, на который можно было бы разместить эту ВМ с выполнением всех ограничений на корректность отображения и запрошенных для него требований в SLA [1, 2]. То есть, в ходе эксплуатации ЦОД возникает фрагментация физических ресурсов, которая приводит к снижению эффективности их использования по критериям: загрузка физических ресурсов ЦОД и процент размещенных запросов из множества поступивших запросов. Фрагментация физических ресурсов может устраняться за счет миграции работающих виртуальных ВМ.

В работе рассмотрены схемы организации “живой” миграции виртуальных машин в рамках одного ЦОД. Указаны гипервизоры, которые поддерживают рассмотренные схемы миграции. Проведено сравнение потенциальных возможностей схем по критериям: время, затрачиваемое на освобождение сервера-источника; общее время миграции; промежуток времени, в течение которого в процессе миграции виртуальная машина недоступна; общее количество переданных данных в процессе миграции и уменьшение производительности мигрирующей виртуальной машины.

2. ЗАДАЧА “ЖИВОЙ” МИГРАЦИИ ВИРТУАЛЬНЫХ МАШИН В ЦОД

Основными факторами, определяющими время миграции, являются:

- пропускная способность выделенного для миграции виртуального канала;
- объем передаваемой информации в байтах во время миграции – зависит от скорости изменения оперативной памяти виртуальной машины.

Для организации “живой” миграции в ЦОД необходимо проложить виртуальный канал между сервером-источником и сервером-приемником таким образом, чтобы он не повлиял на характеристики работающих в ЦОД виртуальных машин. Однако, не всегда возможно проложить канал с такой пропускной способностью, чтобы обеспечить приемлемые затраты времени на миграцию (критерии оценки затрат времени, по которым оценивается эффективность средств миграции приведены в разделе 3). Также возникает проблема поддержания необходимой пропускной способности виртуального канала выделенного для миграции. Так как различные виртуальные каналы могут использовать одни и те же физические каналы связи и одно и то же коммутационное оборудование.

Для ряда виртуальных машин работающих в режиме SaaS возможно получение верхней оцен-

ки скорости изменения памяти, так как для таких виртуальных машин ресурс в основном использует только одно приложение. Для этого потребуется найти историю поведения приложения, для которой количество записей и считываний из памяти максимально.

Для виртуальных машин, работающих в режиме PaaS, может оказаться проблематичным получение верхней оценки скорости изменения памяти. Возможность и метод получения будут зависеть от типа платформы запущенной на виртуальной машине.

Для виртуальных машин, работающих в режиме IaaS получение верхней оценки скорости изменения памяти невозможно. Это связано с тем, что неизвестно какие программы и когда запускаются на виртуальной машине.

3. "ЖИВАЯ" МИГРАЦИЯ В СОВРЕМЕННЫХ ГИПЕРВИЗОРАХ

"Живая" миграция поддерживается во всех современных гипервизорах: KVM [3], Xen [4], VMware [5], Hyper-V [6]. Сравнение эффективности "живой" миграции в различных гипервизорах приведено в статье [7].

"Живая" миграция по сравнению с миграцией остановленной или приостановленной виртуальной машины отличается тем, что во время процесса миграции практически все время виртуальная машина остается доступной. Остановленная виртуальная машина (powered-off): виртуальная машина полностью выключается на сервере-источнике, перемещается, а после включается на сервере-приемнике. Приостановленная виртуальная машина (suspended): состояние всех работающих приложений сохраняется, виртуальная машина переходит в "приостановленное" состояние (аналогично спящему режиму персонального компьютера), далее происходит перемещение виртуальной машины с сервера-источника на сервер-приемник, после этого виртуальная машина вновь запускается с восстановлением состояний всех ранее работающих приложений.

Во всех схемах "живой" миграции используются приостановленные виртуальные машины.

Основные критерии, по которым сравнивают различные схемы миграции:

1. Промежуток времени, после которого сервер-источник освободится (далее eviction time).

2. Общее время миграции (далее total migration time).

3. Промежуток времени, в течение которого в процессе миграции виртуальная машина недоступна (далее downtime).

4. Общее количество переданных данных в процессе миграции виртуальной машины.

5. Уменьшение производительности мигрирующей виртуальной машины (в качестве наиболее объективной оценки уменьшения производительности является увеличение времени выполнения приложений выполняемых на виртуальной машине).

4. СХЕМЫ "ЖИВОЙ" МИГРАЦИИ

"Живая" миграция виртуальных машин включает в себя три этапа [8]:

1. Передача состояния процессора. Передается текущее состояние процессора, информация о размещенных на процессоре в данный момент процессах, используемые в данный момент страницы памяти. Этот этап незначительно влияет на downtime, так как нужно передавать небольшое количество информации.

2. Передача состояния оперативной памяти. Состояние памяти состоит из страниц памяти гостевой операционной системы (т.е. операционной системы виртуальной машины) и страниц памяти всех запущенных процессов внутри виртуальной машины. Некоторые гипервизоры могут отслеживать, какие страницы памяти используются, а какие нет. Это нужно для уменьшения времени миграции, чтобы не передавать лишний раз неиспользованные страницы. Данный этап называется memory migration.

3. Передача содержимого внешнего диска. Это опциональная часть миграции. Диск не нужно мигрировать, если он доступен с сервера-источника и с сервера-получателя, например, в случае Network Attached Storage (NAS). Данная стадия называется storage migration. Если внешний диск не может быть подсоединен к серверу-получателю, то его необходимо перемещать. В этом случае время миграции может значительно увеличиться, так как размер диска может составлять десятки или сотни гигабайт. Так же как и с передачей оперативной памяти, гипервизор может отслеживать, какие блоки данных используются на диске для того, чтобы не передавать неиспользуемые блоки.

Существует две основных схемы организации "живой" миграции: Pre-Copy "живая" миграция [9, 10] и Post-Copy "живая" миграция" [11–13]. В данных схемах рассматриваются первые два этапа миграции. Остальные схемы являются модификацией этих схем.

Pre-Copy "живая" миграция [9, 10]. Оперативная память копируется с сервера-источника на сервер-получатель итерациями, сама виртуальная машина в процессе миграции остается на сервере-источнике. На первой итерации копируется вся память, на последующих итерациях копируются только те страницы, над которыми были произведены изменения. После того как количество итерации превысило некоторый заранее за-

Таблица 1. Сравнение схем организации “живой” миграции

Подход к миграции	Eviction time	Total migration time	Downtime	Transmitted data
Pre-Copy	2	2	1	2
Post-Copy	2	1	3	1
Scatter-Gather	1	2	2	1

данный порог или количество модифицированных страниц на очередной итерации стало меньше заранее заданного порога, виртуальная машина приостанавливается (переводится в состояние suspend) и полностью переносится на сервер-получатель вместе с состоянием процессора и оставшимися модифицированными страницами памяти. Именно этот период неактивности виртуальной машины называется downtime. Для виртуальной машины, которая не интенсивно работает с памятью, downtime будет небольшим, для интенсивно работающих машин во всех итерациях будет передаваться большое количество страниц, когда количество итераций превысит заранее заданный порог, то модифицированных страниц останется все еще много, и поэтому downtime будет большим.

Post-Copy “живая” миграция” [11–13]. Сначала копируется состояние процессора на сервер-получатель. Далее виртуальная машина в процессе миграции уже выполняется на сервере-получателе. Во время миграции сервер-источник посылает страницы памяти на сервер-получатель (эта стадия называется pre-paging) – в надежде, что большинство страниц будет передано на сервер-получатель до того, как виртуальная машина начнет к ним обращаться. Если виртуальная машина обратиться к странице, которой еще нет на сервере-получателе, то будет возбуждено событие remote page fault. В этой ситуации посылается сообщение на сервер-источник с просьбой переслать нужную страницу памяти, далее сервер-источник пересылает ее, сервер-получатель принимает. Чем таких событий будет меньше, тем эффективность этого подхода переноса виртуальной машины будет выше по сравнению с предыдущим подходом. В отличие от подхода, описанного выше, в этом подходе каждая страница памяти передается ровно один раз по сети, что снижает количество использованных сетевых ресурсов. Downtime в этом случае минимальный, так как состояние процессора переносится на первом же шаге на сервер-получатель. Однако деградация производительности сразу после переноса состояния процессора на сервер-получатель может быть больше, чем в Pre-Copy подходе, так как приложения, работающие на виртуальной машине могут быть недоступны до тех пор, пока их рабочие окружения не будут переданы на сервер-получатель.

Гибридная схема. Сначала миграция работает согласно Pre-copy схеме с некоторым порогом на количество итераций. После завершения всех итераций производится передача состояния процессора, далее виртуальная машина уже работает на сервере-получателе. Завершающая стадия миграции работает согласно Post-copy схеме.

Scatter-Gather “живая” миграция [14]. Она основана на Post-Copy схеме. В схеме Scatter-Gather используются посредники. Сначала виртуальная машина переносится с сервера-источника на эти посредники, потом сервер-получатель скачивает с посредников данную виртуальную машину. Посредниками могут выступать другие сервера или middlebox [15]. В описанных выше схемах, значение критерия eviction time равно значению критерия total migration time. Эта схема позволяет снизить значение eviction time во время миграции, если сервер-получатель скачивает виртуальную машину намного медленнее, чем сервер-источник может ее передавать.

Сравнение выше описанных схем организации “живой” миграции приведено в Таблице 1. Данная таблица построена по результатам экспериментов из работы [14]. Для каждой схемы в каждом эксперименте получен ранг от 1 до 3 в соответствии с полученными в работе [14] результатами. 1 – самое малое значение, 3 – самое большое значение. Ранг позволяет понять, какая схема лучше для каждого критерия в среднем по результатам экспериментов.

Однако значения критериев 1–4 характеризующих схему миграции значительно зависят от способа реализации схемы. Влияние на значение этих критериев будет оказывать порядок передачи страниц памяти с сервера-источника на сервер-получатель. Порядок передачи страниц может оказывать сильное влияние на количество передач каждой страницы. Если виртуальная машина делает много обращений к страницам памяти, которые расположены не на том сервере, где выполняется виртуальная машина, то общее количество передач страниц памяти в процессе миграции увеличивается тем сильней, чем больше таких обращений. Вопрос о способе выбора наилучшего порядка передачи страниц памяти является нерешенным.

Также значения критериев 1–4 могут зависеть от способа распределения памяти виртуальной машины и размера страниц. Оптимизация реализации схем миграции по этим характеристикам проблематична. Для такой оптимизации потребу-

ется модификация операционных систем и компиляторов.

При одновременной миграции нескольких виртуальных машин с одного сервера на другой можно уменьшить количество передаваемых данных, используя метод Deduplication migration [14]. Виртуальные машины на одном сервере используют много общих библиотек, так же они могут иметь одну и ту же операционную систему – все это потенциальная возможность уменьшить количество передаваемых страниц. Также существуют способы параллельной миграции, миграции со сжатием данных [14].

5. ПРОБЛЕМА "ЖИВОЙ" МИГРАЦИИ В ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМАХ РЕАЛЬНОГО ВРЕМЕНИ

Вычислительная система реального времени работает в одном из заранее определённых режимов [16]. Например, для комплекса бортового оборудования международной космической станции (МКС) определены следующие режимы работы [17]:

- стандартный режим;
- режим микрогравитации для выполнения научных экспериментов;
- режим сближения и стыковки с транспортными кораблями;
- режим для выхода экипажа в открытый космос;
- режим выживания с отключением наименее важных экспериментов и систем;
- режим аварийного покидания экипажем МКС.

Режим работы характеризуется набором функциональных задач (программ), которые должны выполняться в этом режиме, и набором передаваемых сообщений [16]. Для каждой программы задаются требования к ее выполнению в реальном времени, а для каждого сообщения требования к его передаче в реальном времени.

Требования к выполнению программ в реальном времени могут задаваться одним из способов [18]:

Способ 1: $a_k = (s_k, f_k, t_k)$, где s_k – директивный срок начала выполнения программы k (выполнение программы должно начаться не раньше этого срока); f_k – директивный срок завершения выполнения программы k (выполнение программы должно завершиться до наступления этого срока), t_k – время выполнения программы.

Способ 2: $a_k = (F_k, t_k)$, где F_k – частота ($1/F_k$ – период) выполнения программы; t_k – время выполнения программы. Частота определяет набор отрезков времени выполнения программы, длина которых равна ее периоду.

Второй способ задания требований к выполнению работ в реальном времени может быть сведен к первому. Вычисляется большой цикл как наименьшее общее кратное периодов выполнения работ. Количество запусков работы в большом цикле равно количеству ее периодов в большом цикле. Директивные сроки каждого запуска работы определяются временем начала и завершения соответствующего периода.

Требования к передаче сообщений в реальном времени задаются аналогично требованиям к выполнению программ.

Интерес к подходам организации "живой" миграции в системах реального времени обусловлен требованием "бесшовного" переключения режимов работы системы. При смене режимов работы изменяется набор выполняемых прикладных программ:

– часть программ должна выполняться как в старом, так и новом режиме работы системы,

– часть программ снимается с выполнения,

– добавляются новые программы.

Для каждого режима работы строится свое расписание выполнения программ [16, 18], что бы обеспечить работу системы в реальном времени. В разных режимах работы вычислительной системы реального времени с архитектурой интегрированной модульной авионики [16] программа может выполняться на разных вычислительных модулях. Для программ, которые должны выполняться как в новом, так и в старом режиме требуется обеспечить, что бы во время переключения режима работы системы они не прекращали выполняться или были не доступны не более заданного времени.

6. ЗАКЛЮЧЕНИЕ

Схему для организации "живой" миграции виртуальных машин можно изменять в зависимости от состояний ресурсов в центре обработки данных и от интенсивности работы с памятью мигрирующей виртуальной машины.

На значение критериев эффективности миграции (промежуток времени, после которого сервер-источник освободится; общее время миграции; промежуток времени, в течение которого в процессе миграции виртуальная машина недоступна; общее количество переданных данных в процессе миграции; уменьшение производительности мигрирующей виртуальной машины) кроме выбранной схемы миграции оказывает влияние способ реализации схемы. Сильное влияние может оказать алгоритм, определяющий порядок передачи страниц виртуальной машины с сервера-источника на сервер-получатель. Проблемы построения этого алгоритма связаны с проблемами построения алгоритмов кэширования. "Уровень попаданий" в кэш означает то, насколько велики затраты времени на считывание и запись данных в память. "Уровень попаданий" в страницы памяти, которые размещены на одном сервере

с процессором виртуальной машины, также определяет скорость обращения к памяти и, кроме того, объем данных передаваемых в процессе миграции.

В работе [19] было исследовано влияние “живой” миграции на эффективность работы ЦОД по критериям загрузки физических ресурсов и процента размещенных запросов из исходно поступившего набора запросов на создание виртуальных сетей. Если миграция допустима, то повышается загрузка ресурсов и увеличивается процент размещенных запросов. Однако миграция виртуальных ресурсов без их останова может значительно увеличивать нагрузку на сетевые ресурсы ЦОД. Для миграции виртуальной машины нельзя выделить слишком много сетевых ресурсов без ухудшения характеристик других работающих виртуальных машин, поэтому может потребоваться значительное время для миграции перемещаемых виртуальных машин. Это может приводить к тому, что время размещения новых виртуальных машин будет неприемлемо большим.

7. БЛАГОДАРНОСТИ

Работа выполнена при финансовой поддержке РФФИ, грант № 19-07-00614.

СПИСОК ЛИТЕРАТУРЫ

1. *Костенко В.А., Чупахин А.А.* Подходы к повышению эффективности эксплуатации ЦОД. Программирование. 2019. № 5. С. 36–42.
2. *Зотов И.А., Костенко В.А.* Алгоритм распределения ресурсов в центрах обработки данных с единым планировщиком для различных типов ресурсов. Известия РАН. Теория и системы управления. 2015. № 1. С. 61–71.
3. Kernel Virtual Machine. https://www.linux-kvm.org/page/Main_Page
4. Xen. https://www.xenproject.org/
5. VMware. https://www.vmware.com/
6. Hyper-V. https://docs.microsoft.com/en-us/virtualization/index#pivot=main&panel=containers
7. *Hu W., Hicks A., Zhang L., Dow E.M., Soni V., Jiang H., Bull R., Matthews J.N.* A quantitative study of virtual machine live migration. Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference. ACM, 2013.
8. *Soni G., Kalra M.* Comparative Study of Live Virtual Machine Migration Techniques in Cloud. International Journal of Computer Applications. 2013. V. 84. № 14.
9. *Clark C., Fraser K., Hand S., Hansen J., Jul E., Limpach C., Pratt I., Warfield A.* Live migration of virtual machines. Proceedings Netw. Syst. Des. and Implementation. 2005. P. 273–286.
10. *Nelson M., Lim B.H., Hutchins G.* Fast transparent migration for virtual machines. Proceedings USENIX Ann. Tech. Conf. 2005. P. 25.
11. *Hines M.R., Deshpande U., Gopalan K.* Post-copy live migration of virtual machines. SIGOPS Operating Syst. Rev. 2009. V. 43. № 3. P. 14–26.
12. *Hirofuchi T., Yamahata I.* Yabusame: Postcopy live migration for Qemu/KVM. Presented at KVM Forum, 2011.
13. *Lagar-Cavilla H., Whitney J., Scannell A., Patchin P., Rumble S., De Lara E., Brudno M., Satyanarayanan M.* SnowFlock: Rapid virtual machine cloning for cloud computing. Proceedings EuroSys: Fourth ACM Eur. Conf. Comput. Syst. 2009. P. 1–12.
14. *Deshpande U., Chan D., Chan S., Gopalan K., Bila N.* Scatter-Gather live migration of virtual machines. IEEE Transactions on Cloud Computing, January-March. 2018. V. 6. № 1.
15. Middlebox. http://yuba.stanford.edu/~huangty/sigcomm15_preview/mbpreview.pdf
16. *Костенко В.А.* Архитектура программно-аппаратных комплексов бортового оборудования // Изв. вузов. Приборостроение. 2017. Т. 60. № 3. С. 229–233.
17. *Куминов В., Наумов Б.* Космические компьютеры: открытые стандарты и технологии выходят в открытый космос. Мир компьютерной автоматизации. 2002. № 3. С. 71–79.
18. *Костенко В.А., Смирнов А.С.* Потоковые алгоритмы планирования вычислений в интегрированной модульной авионике. Известия РАН. Теория и системы управления. 2019. № 3. С. 104–114.
19. *Костенко В.А., Чупахин А.А.* Влияние миграции на эффективность использования ресурсов центров обработки данных. Программные системы и инструменты. Тематический сборник. Т. 17. МАКС Пресс Москва, 2017. С. 85–91.

ПРОГРАММИРОВАНИЕ, 2020, № 5, с. 16–24

ПРОГРАММНАЯ ИНЖЕНЕРИЯ, ТЕСТИРОВАНИЕ И ВЕРИФИКАЦИЯ ПРОГРАММ

УДК 004.5

ПОДХОДЫ К РАЗРАБОТКЕ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА

© 2020 г. В. Н. Лукин[a],*, А. Л. Дзюбенко[b],, Ю. Б. Чечиков[a,b],*****

[a] *Московский авиационный институт (национальный исследовательский университет)*
125993 Москва, Волоколамское шоссе, д. 4, Россия

[b] *Финансовый университет при Правительстве Российской Федерации*
125167 Москва, Ленинградский пр-т, д. 49, Россия

**E-mail: lukinvn@list.ru*

***E-mail: al_dz@list.ru*

****E-mail: yourych@mail.ru*

Поступила в редакцию 29.07.2019 г.
После доработки 01.10.2019 г.
Принята к публикации 01.11.2019 г.

В работе рассматриваются проблемы, связанные с современным пользовательским интерфейсом. Проводится анализ интерфейсов, построенных на парадигме реализации, и выявляются ее слабые стороны. Оцениваются существующие методологии разработки интерфейсов. Приводится обоснование использования метафорического или идиоматического подхода при создании (усовершенствовании) пользовательского интерфейса.

На ежегодном собрании Кеннет Олсен, инженер, основавший компанию Digital Equipment Corp. и в настоящее время руководящий ею, признался, что не знает, как приготовить кофе с помощью микроволновой печи компании.

DOI: 10.31857/S0132347420050052

1. ВВЕДЕНИЕ

Статья посвящена проблемам пользовательского интерфейса. Для того, чтобы программный продукт был востребован, он должен выполнять не только свои функции согласно техническому заданию [1], но и быть удобным и простым в использовании. Для этого должен быть тщательно спроектирован и реализован пользовательский интерфейс.

В распоряжении дизайнера интерфейса имеются разные подходы и методологии, но, наиболее проигрышная, на наш взгляд, методология, основанная на парадигме реализации.

Наиболее удачным, с точки зрения разработки комфортного пользовательского интерфейса, выглядит когнитивный подход, в рамках которого появились метафорическая и идиоматическая парадигмы. Они наиболее полно учитывают психологические и технологические требования пользователей, предъявляемые к интерфейсам, и именно в этом направлении целесообразно проводить дальнейшие исследования.

Создание удачного интерфейса требует изменения технологии разработки программного продукта, с учетом приоритетности качества взаимодействия, понимания важности комфорта пользователя руководством проекта. Следует опираться на результаты научных исследований [2–6] и на успешный практический опыт создания пользовательского интерфейса в области разработки прикладных программ, закрепленный отчасти в национальных стандартах [7–24].

Хорошие результаты могут получиться, если разработка пользовательского интерфейса, хотя бы на уровне макета, начинается сразу после анализа требований к системе, лучше до разработки программы, в крайнем случае, одновременно. Лучше, если в группу разработчиков включить профессиональных специалистов (не программистов!) [25], а, также, представителя заказчика, обладающего высокой квалификацией и административным ресурсом, чтобы вовремя высказать замечания и внести исправления в проект. На уровне технического задания необходимо детально прописать требования к интерфейсу. Это, помимо прочего, позволит более точно оценить время и стоимость разработки. Здесь же следует определить категории и количество групп пользователей, их цели при обработке информации. И, на протяжении всей работы по созданию интерфейса, необходимо обеспечить доминирова-

ние интересов пользователя над интересами программистов.

2. ИССЛЕДОВАНИЕ ВОПРОСА

Конкурентная борьба за увеличение продаж персональных компьютеров потребовала не только усовершенствования технических характеристик компьютеров и создания все более мощных операционных систем, но и разработки более совершенных средств взаимодействия пользователей и компьютеров. И, если на раннем этапе развития компьютерной индустрии, интерфейс предназначался для работы специально подготовленного инженерно-технического персонала, то в настоящее время, подавляющее число пользователей взаимодействует с вычислительным устройством, выполняя обычную работу. Таким образом, при проектировании интерфейса становится необходимостью учет интересов рядового пользователя [2, 7, 8, 26–28].

В связи с тем, что взаимодействие с персональным компьютером стало органичной частью работы пользователей, работающих в различных предметных областях, решение задач проектирования и реализации пользовательского интерфейса потребовало усилий специально подготовленных групп профессионалов. Соответственно, в крупных компаниях-разработчиках вычислительной техники и программного обеспечения появились подразделения, специализирующиеся на анализе потребностей пользователей программного обеспечения и создании эргономичного дизайна для решения ими своих профессиональных задач [10, 19]. Естественно, в ходе этих разработок возникает актуальная необходимость найти ответы на вопросы: “Как организовать взаимодействие пользователя с программными приложениями, чтобы он не ушел к конкуренту за более легким и удобным способом общения с компьютером? По каким критериям можно оценить качество интерфейса, и с каких позиций надо подходить к разработке успешного интерфейса?”

В период, когда в самом разгаре был инженерный подход к разработке программных систем, и программные комплексы разрабатывались в стиле индустриального материального производства, основными пользователями были операторы (мы не рассматриваем пласт научных и расчетных программ, пользователями которых были сами авторы). Это специалисты, обученные работе с аппаратно-программным обеспечением, они “прогоняли” программы согласно прилагаемым инструкциям, в которых расписывался сценарий деятельности. В некотором смысле, они были элементом технологического процесса, как рабочие в цеху или доярки на ферме. Разумеется, никому в голову не приходило особо заботиться о форме представления диалога (интерфейса), нужно было только для минимизации ошибок оператора заботиться об однозначности и корректности инструкции.

Таким образом, инженерно-технический (Machine-Centered) подход рассматривал процесс разработки с точки зрения функциональных возможностей компьютера, “изнутри-вовне”, когда сначала разрабатывается алгоритм, пишутся программы, затем проектируется ввод/вывод. По умолчанию предполагалось, что процесс решения задачи человеком подобен работе компьютера, только роль программы играла инструкция, которую следовало неукоснительно выполнять. Для достижения цели большая задача разбивалась на подцели, которые, при необходимости, также разбивались на подцели.

Этому подходу способствовало то, что тогда процесс решения сильно ограничивался техническими возможностями компьютеров, которые имели малый объем памяти и недостаточное быстродействие, так что все внимание разработчиков ЭВМ и программистов концентрировалось на усовершенствовании машин и повышении эффективности программ. Ресурсная эффективность считалась важнейшей характеристикой качества прикладной программы, поэтому из техники старались выжать все, что можно, и только потом принимались во внимание и учитывались адаптивные возможности человека. Надо сказать, они были достаточно высокими для компенсации слабого сопряжения человек-машина. Недостаток внимания к проблемам пользовательского интерфейса объясняется особенностью тогдашнего контингента пользователей, среди которых не было не подготовленных специально людей.

Программисты того времени неплохо разбирались в электронной технике, прекрасно ориентировались в существующем системном программном обеспечении и, волей-неволей, вырабатывали стиль профессионального мышления, сходный с алгоритмическим (стиль “центрального процессора”). Привычка мыслить в терминах машины помогала расшифровать специфические коды, играющие роль сообщения об ошибках: нередко это была комбинация неоновых лампочек, горящих на пульте. Понятно, они требовали такого же подхода и от операторов. Таким образом, инженерно-технический подход к созданию пользовательского интерфейса был основан на предположении, что человек работает с компьютером подобно самому компьютеру, следовательно, пользователь, независимо от профиля своей работы, должен думать как разработчик, а интерфейс, соответственно, ориентировался именно на функциональные характеристики программы.

Подобный подход и привел к созданию такой парадигмы разработки пользовательского интерфейса, как парадигма реализации, при которой перед пользователем стоит непростая задача:

разобраться в интерфейсе, не владея логикой построения программы.

Но, начиная с 1970-х годов, бурное развитие автоматизированных систем, в контуре управления которыми находился человек-оператор, показало, что игнорирование психофизиологических характеристик человека резко снижает эффективность систем и даже может быть причиной техногенных катастроф [21, 30–32].

Например, в середине 80-х годов была предпринята попытка автоматизировать одну из ведомственных поликлиник Мосгорисполкома. Работа была поручена крупному предприятию оборонного комплекса. Техническая сторона проекта была сделана грамотно: продумана архитектура сети, конфигурация рабочих мест врачей и медсестер, продублированы особо важные элементы системы.

С целью экономии народных средств, было решено не оснащать каждое рабочее место врача принтером, а поставить один качественный высокоскоростной принтер в холле каждого этажа. Именно это решение привело к тому, что, спустя месяц после запуска, система встала: медперсонал заявил, что отказывается от ее услуг, т.к. бегать за каждой бумажкой в холл и обратно невозможно, нет гарантии, что твой документ не унес кто-то другой и, поэтому, дешевле работать по старой ручной технологии.

Систему, конечно, переделали как положено, установив принтер на каждое рабочее место, но этот пример наглядно показывает, что даже технически грамотные, но не учитывающие интересы пользователей системы нежизнеспособны.

Таким образом, раз модель использования отражает подробности реализации программы в коде, понятно, что никто, кроме самого разработчика, не сможет ориентироваться в ней лучше него. Конструкция интерфейса полностью соответствует конструкции программы: по одной кнопке на каждую функцию, каждому модулю кода соответствует свое окно, внутренние структуры данных и алгоритмы порождают команды и процессы обработки данных пользователя.

Чтобы квалифицированно пользоваться таким интерфейсом, пользователь не только должен обладать профессиональными знаниями о том, как программа устроена, он должен думать, как программист и, фактически, быть программистом, что просто нереально. Получается, что таким образом интерфейс проектируется только для программистов.

Естественно, никто не обвиняет программистов в преднамеренном желании создавать сложные для использования интерфейсы. Существует достаточно причин объективного и субъективного характера, из-за которых программисты проектируют интерфейсы, сложные именно для неспециалистов.

Во-первых, это диктуют формулировки технического задания (ТЗ). В нем четко прописано, какие именно функции должна выполнять система. Для удобства ее сдачи заказчику программисту проще каждую функцию оформлять в виде кнопки или окна: тогда сразу видно, что конкретная функция реализована. Естественно, строить систему с интерфейсом, ориентированным на реализацию, проще всего: при создании новой функции добавляется фрагмент интерфейса, отражающий данную функцию. Если что-то не работает, легко выяснить, в чем причина, и быстро исправить ошибку.

Что касается требований ТЗ к интерфейсу, то, как правило, в ТЗ в списке требований к системе стоит фраза: “Система должна иметь удобный и эргономичный интерфейс”. А что конкретно стоит за этим требованием и как заказчик будет проверять его выполнение – неизвестно. Поэтому заказчик вынужден принимать тот интерфейс, который ему предложили, и принято считать, что этого достаточно.

Во-вторых, как показывает опыт разработки программных проектов, в большинстве случаев плановые сроки разработки системы нарушаются, и для того, чтобы сдать заказчику систему в срок, все силы и внимание в первую очередь направляются на разработку функционала, а не на создание удобного интерфейса, но с неполным функционалом.

Если при сдаче системы приоритетнее функционал, разработчики рассчитывают, что претензии к интерфейсу можно будет отнести к категории замечаний, которые выявились в последний момент и не были оговорены заранее, и есть возможность уговорить заказчика перенести усовершенствование интерфейса на более поздний период, когда будут исправляться выявленные ошибки. Но здесь может таиться серьезная опасность для системы. Если интерфейс привязан к функционалу, систему заказчику сдать можно, особенно если систему принимают не потенциальные пользователи, которые с ней потом будут работать. Но, если в приемке системы участвует контингент, заинтересованный в качестве взаимодействия, то для соответствия реальной технологии работы потребуется серьезное изменение интерфейса. Реализация замечаний может привести к изменению структуры информационной базы системы, что, в свою очередь, потребует изменения алгоритмов обработки информации и даже структуры всей системы в целом. А это уже практически создание новой системы, что по срокам и трудоемкости не идет ни в какое сравнение с исправлением ошибок. Поэтому, скорее всего, интерфейс немножко подправят, но его суть останется прежней – это будет интерфейс, основанный на реализации.

В-третьих, при проектировании имеет большое значение субъективная система ценностей разработчика. Самый важный и принципиальный вопрос, на который должен явно ответить разработчик – чьи интересы важнее, пользователя или разработчика. Будет ли разработчик приближать программные продукты к образу мышления пользователей или заставит пользователей подстраиваться под чуждую им машинную логику, облегчая себе жизнь? Как правило, разработчик облегчит жизнь себе, особенно, если нужно успеть сдать систему в срок. Конечно, интересы пользователей оказываются на последнем месте, и в результате неудобный интерфейс готов.

В-четвертых, играет роль структура и состав коллектива разработчиков программного продукта. Если руководитель проекта учитывает важность удобного интерфейса для пользователя, то, для нивелирования субъективной системы ценностей отдельных программистов, в коллектив должен быть включен специалист по дизайну интерфейса, обязанность которого – обеспечить качество взаимодействия с пользователями. Он должен иметь право блокировать выпуск некачественного интерфейса, что позволит защитить интересы пользователя в конфликтных ситуациях с программистами. Однако, подобные решения в результате могут не только повысить стоимость проекта, но и увеличить конфликтность в команде. Тем не менее, руководитель проекта должен обладать административной мудростью и управленческой твердостью, чтобы с самого начала всесторонне учитывать интересы пользователя и снижать напряженность в коллективе. К сожалению, такие руководители встречаются нечасто.

В-пятых, для обеспечения надежности изделия инженерам всегда нужно знать, как именно оно устроено и работает, поэтому парадигма реализации, которая помогает понять, что происходит внутри системы, их устраивает. Тот факт, что это неоправданно усложняют жизнь пользователей, отходит на второй план и кажется разработчикам несущественным побочным эффектом.

Критика данного подхода стала появляться в литературе с середины 80-х годов, когда накопился мировой опыт использования ЭВМ в различных сферах и появились осознанные рекомендации по выбору тех или иных интерфейсных решений [29, 32, 33]. Эти вопросы рассматривались такими исследователями, как Б. Шнейдерман [34], М. Минанси [35], Д. Норман [3], Гультяев А. [36]. В результате этих исследований пришло понимание того, что интересы пользователя и программиста часто противоречат друг другу, и, если в рамках программного проекта этой проблемой не заниматься, то программист решит ее удобным для себя способом.

К концу 80-х–началу 90-х годов в области проектирования пользовательских интерфейсов назрел методологический кризис, связанный с отсутствием ясного, единого понимания технологии их создания. Подход, основанный на теории деятельности, позволил проектировщикам найти путь из этого кризиса. Система “человек-компьютер” рассматривается в нем как комплекс деятельностных понятий и представлений. Этот подход (Activity-Centered Design, ACD) [11, 12, 28] близок к инженерно-технической парадигме реализации, но в нем учитываются интересы пользователей. Теория деятельности, лежащая в основе этого подхода, представляет компьютер уже в качестве инструмента, с помощью которого человек решает различные задачи, и здесь именно деятельность человека влияет на интерфейс.

Согласно принципам теории деятельности, выдвинутой советским психологом А.Н. Леонтьевым[1], человек рассматривается через призму того, как он взаимодействует с окружающим миром, и весь поток активности пользователя раскладывается на последовательность связанных задач и подзадач, логические этапы. Этот подход позволяет анализировать цели, внешние и внутренние задачи, порядок и вид операций пользователя, совершаемых для достижения итогового результата, и, по результатам анализа, разработать интерфейс, наиболее подходящий для данного вида деятельности.

В иерархии ACD деятельность состоит из задач, задачи состоят из действий, а действия составлены из операций. Такая структура упрощает программистам жизнь, так как, в результате, систему можно представить в виде набора функций, правда, уже не программных, а технологических, связанных с непосредственной деятельностью пользователей.

Схема ACD особо подчеркивает важность контекста пользователя. Метод ACD полезен при разделении на составные части того, что делает пользователь, но этот метод не отвечает на важный вопрос: почему пользователь приступает к этой активности, задаче, действию или операции? Он просто отражает технологию работы пользователя без анализа целей, которые преследует пользователь на каждом этапе работы.

Существенный недостаток подхода на основе анализа деятельности – невозможность анализировать сложные умственные виды деятельности пользователя, но более опасным видится риск повторить, а не модифицировать устаревшую ручную технологию работы пользователей при автоматизации предметной области [5, 37].

[1] А.Н. Леонтьев, Становление психологии деятельности, Москва, НПФ “Смысл”, 2003, 880 стр., илл.

Помимо прочего, стоит отметить, что системы с неудобным для заказчика интерфейсом имеют короткий жизненный цикл, а это проигрышно как для заказчика, так и для разработчика. Поэтому от подходов, приводящих к созданию неудобных для заказчика интерфейсов рациональнее отказываться.

Можно подумать, что в условиях, когда не было не только графического, но и даже цветного дисплея, добиться удобства взаимодействия просто невозможно. Но это не так. В середине 1980-х годов разрабатывалось довольно много систем с интерфейсом не только в инженерном стиле, но и в стиле деятельности, хотя достаточной теоретической базы еще не было. Некоторые из них были довольно удачными, для этого необходимо было внимательно изучить технологию работы потенциального пользователя и сделать так, чтобы ему было удобно. Так, в частности, разрабатывался интерфейс для системы Биолаб, предназначенной для автоматизации биохимической лаборатории Центральной клинической больницы [38]. Стиль взаимодействия был диалоговый, был реализован удобный и понятный пользователям язык общения с системой, отражающий все технологические процессы выполнения исследований, расчетов, отображения данных, выдачи документов. И, возможно, поэтому жизненный цикл системы длился 14 лет.

Но, даже в лучшем варианте, подходы со стороны программы или даже технологии, не дают пользователю достаточно свободы и не учитывают того, что называют психоэмоциональным комфортом пользователей [1, 13, 33, 39–41]. Возможно, для подготовленных и достаточно квалифицированных специалистов это не так важно, но при массовом использовании вычислительных средств на таких пользователей рассчитывать не приходится. Поэтому были предложены различные подходы к созданию пользовательского интерфейса, учитывающие этот самый комфорт [3, 4, 22, 34].

Что же конкретно предлагается взамен? Альтернатива – когнитивный подход или подход “извне-вовнутрь”, пришедший на смену алгоритмическому моделированию. Этот подход рассматривает пользователя как центральную фигуру процесса взаимодействия с системой. Конечно, такая возможность появилась с массовым производством персональных компьютеров, когда технические ограничения перестали быть определяющими, и можно было более полно учитывать психологические особенности человека и создавать более комфортные системы, учитывающие интересы пользователя – непрофессионала в области вычислительной техники. Ожидать от этих пользователей стремления адаптироваться под технику стало бессмысленным.

Действительно, пока компьютер был диковинкой, люди психологически были готовы адаптироваться к нему, а теперь появились все основания считать, что машину надо приспосабливать к человеку с помощью более тщательно сконструированного интерфейса.

Разработку, в таком случае, естественнее проводить по пути “извне-вовнутрь”. Извне находятся человек, дисплей, выводимая информация. Внутри – программа, алгоритм. Это подход ориентирован на пользователя, в настоящее время он господствующий. В общем случае, сначала определяются функции будущей системы, описывается диалог пользователя, готовится макет интерфейса, а уже потом под него пишется программа.

Рассматривая процессы и закономерности восприятия, переработки информации и принятия решения, когнитивная психология выявила факторы, определяющие успешность выполнения задачи оператором. И это оказались не функциональные характеристики системы, как предполагалось инженерами раньше, а качество предоставления и управления информацией с точки зрения возможностей и ограничений человека.

Ориентация на характеристики пользователя, исследование способностей к восприятию, когнитивных возможностей и ограничений человека, позволили выявить закономерности взаимодействия человека с автоматизированной системой. Однако, как оказалось, анализа только процессов восприятия и переработки информации человеком недостаточно для проектирования эргономичного интерфейса, поскольку он не позволяет определить состав и последовательность выводимой на экран информации.

В связи с этим, различные исследователи и организации-разработчики программного обеспечения предлагают для разработки интерфейса разные подходы. Так, появились методологии дизайна пользовательского интерфейса, основанные на когнитивном подходе, в частности, целеориентированный дизайн (Goal-oriented design) [37], дизайн, ориентированный на пользователя (User-Centered Design (UCD)) [14], эмоциональный дизайн [42, 44, 34] (рис. 1).

Рассмотрим эти методологии подробнее.

1. Целеориентированный дизайн (Goal-oriented design).

Эта методология разработки пользовательских интерфейсов предложена Аланом Купером [37], она основана на предположении о том, что тщательное изучение целей пользователя и понимание того, к чему он стремится, позволяет понять смысл его деятельности и, таким образом, создать более уместные и качественные продукты.

Цели побуждают людей вести некую деятельность; понимание целей позволяет понять ожидания и устремления пользователей, что, в свою

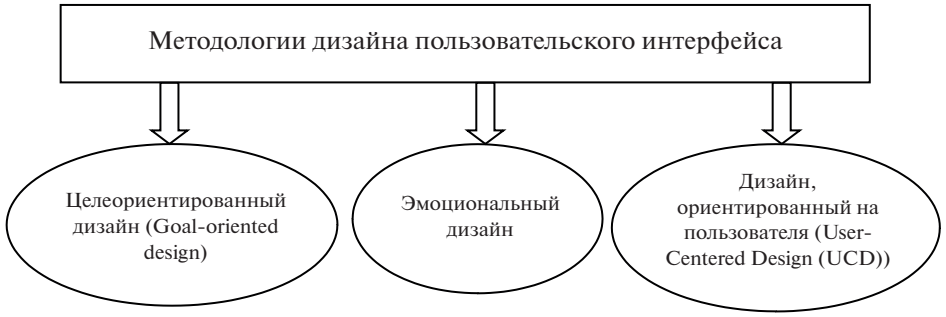

Рис. 1. Схема методологии дизайна пользовательского интерфейса.

очередь, помогает в определении видов деятельности, имеющих реальное отношение к дизайну интерфейса.

Цели определяются человеческими мотивами и, поэтому, со временем не меняются или меняются весьма незначительно. Деятельность и задачи преходящи, поскольку почти целиком основаны на имеющейся в данный момент технологии.

2. Дизайн, ориентированный на пользователя User-Centered Design (UCD) [14].

Суть этой методологии сводится к изучению потребностей и возможностей конечных пользователей и адаптации продукта под их нужды.

Принципы UCD:

• определение целевой аудитории – центральный этап разработки;

• понимание психологии пользователей, видение дизайна через призму взгляда представителей целевой аудитории;

• анализ информации о конкурентах;

• эволюционный дизайн, построенный на обратной связи с пользователями, которая поддерживается с помощью тестирования многочисленных прототипов и управляет процессом разработки;

Принципы UCD успешно реализуются на практике только при наличии постоянной тесной связи с целевой аудиторией. Как только совместно с заказчиком станет ясно, кто будет пользоваться продуктом, можно переходить к выяснению:

1) что представители целевой аудитории хотят от продукта?

2) что он должен делать для них?

3) в каких условиях они будут его использовать?

4) какие функции аналогичных продуктов конкурентов представители целевой аудитории используют чаще всего?

5) чего им не хватает в продуктах конкурентов?

На основе собранных входных данных проектировщики интерфейсов создают прототип, по которому дизайнеры разрабатывают предварительную версию. Версия тестируется представителями целевой аудитории. Их попытки выполнить пользовательские задачи с помощью прототипа, результаты, отзывы и впечатления ложатся в основу разработки. Затем проводится тестирование нового варианта, и, таким образом, процесс продолжается до того момента, пока рациональные предложения пользователей и их серьезные критические замечания не иссякнут.

Другими словами, это концепция создания программных продуктов, которыми люди хотели бы пользоваться.

3. В результате накопленного опыта разработки пользовательских интерфейсов появилась философия эмоционального дизайна, и произошло это в начале семидесятых годов прошлого века [42, 43]. Как показывает практика, зачастую, незначительные и, на первый взгляд, незаметные детали визуального дизайна вызывают у пользователей больший эмоциональный отклик, чем концепция системы в целом. Такого рода детали придают продукту неповторимую индивидуальность и помогают установить с пользователями связь на личностном уровне. Этот прием получил название “эмоциональный дизайн”.

Термин “эмоциональный дизайн” был введен (среди прочих) Аароном Уолтером[2]. В своей книге “Эмоциональный веб-дизайн” он описывает эмоциональный дизайн, используя иерархию человеческих потребностей А. Маслоу.

Воплощение философии эмоционального дизайна принадлежит японскому профессору Митсуо Нагамачи (Mitsuo Nagamachi), который предложил преобразовать эмоциональные переживания и ощущения человека в конкретные свойства продукта. В начале 90-х годов этот подход был применен при создании уникального японского автомобиля Mazda MX-5 Miata.

Впервые в разработке автомобиля на первое место поставили не технические характеристики (скорость разгона или количество лошадиных сил), а чувства, которые испытывает водитель автомобиля – нравится ли ему звук мотора, получа-

[2] http://aarronwalter.com/ ОРИГИНАЛЬНАЯ СТАТЬЯ: “THE PERSONALITY LAYER”.

ет ли он удовольствие от езды, от общения с машиной. Этот подход назвали кансей инжиниринг (kansei engineering) и он с успехом применяется во многих компаниях по всему миру в разных промышленных областях.

Таким образом, при создании продуктов важно учитывать не только конечные цели пользователей (ради чего они покупают продукт), но и их эмоциональные цели.

Эмоциональные цели выражают то, как человек хочет себя чувствовать, работая с продуктом. Когда продукт заставляет пользователей чувствовать себя глупо или неудобно, их самоуважение и производительность труда снижаются, а недовольство растет. Стоит чуть переборщить с таким отношением к пользователям – и они воспользуются первым же шансом, чтобы избавиться от этого продукта. Такой продукт потерпит неудачу – независимо от того, насколько хорошо он позволяет пользователям достигать всех прочих целей. В одной крупной организации, которая занималась анализом и прогнозированием загрузки авиарейсов в России, для этих целей была разработана удобная программная система. Со временем ее “ДОСовский” интерфейс показался одному из авторов устаревшим, и он заменил его современным “виндоувским”. Но пользователи отреагировали крайне отрицательно: у них стали сильно уставать глаза. Пришлось потратить время на проектирование интерфейса, адаптированного к удобной и не утомительной работе пользователя в течение рабочего дня.

Итак, делаем вывод, что при разработке программы, необходимо привести весь проект в соответствие с общими законами психологии, а не бездумно стремиться соответствовать так называемым общепринятым “промышленным стандартам”, которые, отчасти, построены без учета закономерностей мышления и поведения человека (например, раздвоение фокуса внимания, когда сообщение об ошибке и место, где произошла ошибка, находятся в разных частях экрана) [3].

Большая часть проблем, связанных с использованием компьютеров и подобных устройств, возникает, скорее, из-за низкого качества интерфейса, чем из-за сложности самой задачи или же недостатка старания или умственных способностей у пользователя [3, 25, 30, 35, 44–46].

Высшей степенью удобства считается ситуация, когда пользователь ощутил потребность в каком-либо ресурсе или сервисе и тут же появилась простая возможность легко эту потребность удовлетворить.

В результате эволюции когнитивного подхода появились еще две парадигмы интерфейсов: метафорическая и идиоматическая.

Метафорические интерфейсы основаны на интуитивных связях, которые пользователь устанавливает между визуальными элементами интерфейса и его функциональностью. Основная идеология этой парадигмы была заложена фирмой Xerox в конце 60-х годов прошлого века, а расцвет ее связан компьютерами компании Apple и появлением операционной системы Windows компании Microsoft.

По сравнению с парадигмой реализации, здесь пользователю не нужно разбираться в программировании. Пользователи применяют свою интуицию, чтобы через метафору понять предназначение объекта, а разработчики через метафоры структурируют интерфейс по понятным пользователям аналогиям.

Метафорическая парадигма была большим шагом вперед в построении интерфейсов, имеет массу достоинств и широко распространена в настоящее время. Но и здесь возникли проблемы: проявились ограничения и риски использования метафор. При увеличении количества объектов метафоры становятся неудобными, они с трудом подлежат масштабированию, для некоторых программных операций не удается найти метафору из реального мира, например, для изменения формата, а некоторые метафоры со временем перестают быть метафорами (изображение трехдюймовой дискеты для записи сейчас далеко не всем понятно).

Ответом на ограничения парадигм метафорической и реализации стало появление идиоматической парадигмы, которая сейчас развивается и широко исследуется.

Идиома – это свойственное только данному языку устойчивое словосочетание, значение которого не определяется значением входящих в него слов, взятых по отдельности (“собаку съесть”, “пускать пыль в глаза”, “залить на сервер”). Смысл идиомы нельзя понять путем анализа или интуитивно. У идиом нет аналогов в реальном мире, и, поэтому, сначала необходимо узнать смысл идиомы, выучить ее, и только потом использовать ее дальше. Многие известные нам элементы метафорического интерфейса на самом деле являются идиомами. Например, компьютерная мышь – это идиома, хотя ничто в реальной мыши не подсказывает нам, как надо использовать компьютерную мышь, и к тому же, в старых лингафонных кабинетах устройство, похожее на компьютерную мышь, было микрофоном. Но если один раз показать, как пользоваться компьютерной мышью – проблемы отпадут сами собой.

3. ЗАКЛЮЧЕНИЕ

Проблема создания пользовательского интерфейса, зародившись на заре компьютерной эры, продолжает оставаться на удивление актуальной. Нельзя не отметить общее повышение комфортности интерфейса, что связано и с развитием тех-

ники, и с пониманием запросов пользователей, и с развитием методов проектирования. Но жизнь идет вперед, появляются новые коммуникационные возможности и с ними – новые проблемы.

В заключение, хотелось бы привести слова известного специалиста в области разработки пользовательских интерфейсов компании Apple Алана Купера: “Без сомнения, все эти решения потребуют от программистов большей работы. Это нисколько меня не волнует. Я не хочу, чтобы программисты больше работали, но, выбирая из сложности работы программистов и сложности работы пользователей, я немедленно заставляю программистов работать. Работа программиста – удовлетворить пользователя, а не наоборот”.

СПИСОК ЛИТЕРАТУРЫ

1. *Андреев В.Н.* Примерное содержание технического задания по разработке пользовательского интерфейса и тезисов по ведению переговоров // Интернет-источник http://www.usability.ru/toader/articles.htm Центр практичных программ.
2. *Грачев Н.Н.* Психология инженерного труда. М.: Высшая школа, 1998. 331 с., рекомендация Минвуза.
3. Норман Д. Дизайн привычных вещей: Пер. с англ. М.: Издательский дом “Вильямс”, 2006. 384 с.: ил. Парал. тит. англ.
4. Основы инженерной психологии. М.: Высшая школа, 1986. 270 с., учебное пособие, рекомендация Минвуза.
5. *Сугак Екатерина*. Автореферат диссертации на соискание ученой степени кандидата психологических наук (специальность 19.00.03-психология труда, инженерная психология) на тему “Эргономические аспекты проектирования пользовательского интерфейса”.
6. Хрестоматия по инженерной психологии /Сост.: Б.А. Душков, Б.Ф. Ломов, Б.А. Смирнов / Под ред. Б.А. Душкова. Уч. пособие. М.: Высшая школа, 1991. 287 с.
7. ISO 9241-10-1996 Ergonomic requirements for office work with visual display terminals (VDTs). P. 10. Dialogue principies.
8. ISO 9241-12-1998 Ergonomic requirements for office work with visual display terminals (VDTs). P. 12. Presentation of information.
9. ISO 9241-16-1998 Ergonomic requirements for office work with visual display terminals (VDTs). P. 16. Direct manipulation dialogues.
10. *Арлов Л.* Как создать хороший интерфейс пользователя? // Интернет-источник http://www.usability.ru/toader/articles.htm Центр практичных программ.
11. *Батенькина О.В.* Дизайн пользовательского интерфейса информационных систем: учеб. пособие. Омск: Изд-во ОмГТУ, 2014. 112 с.
12. *Гарретт Дж.* Веб-дизайн: книга Джесса Гарретта. Элементы опыта взаимодействия. Пер. с англ. СПб.: Символ-Плюс, 2008. 192 с.: ил.
13. *Герасимов Ю.* Улетный интерфейс // Интернет-источник
14. *Головач В.* Книги и статьи о пользовательских интерфейсах Электронный ресурс. [2018]. Режим доступа: http://www.usethics.ru/lib
15. *Головач В.* Юзабилити-тестирование по дешевке. Электронный ресурс. [2018]. Режим доступа: https://medium.com/usethics-doc/юзабилити-тестирование-по-дешевке-2e853250960f
16. *Головач В.В.* Дизайн пользовательского интерфейса // Интернет-источник http://www.uibook1.ru, 146 с. Центр практичных программ
17. ГОСТ Р 56274-2014. Общие показатели и требования в эргономике. Национальный стандарт Российской Федерации: изд. офиц.: введен 2016-01-01 / НИИ экономики связи и информатики “Интерэкомс”. Москва: Стандартинформ, 2015. IV. 26 с.
18. ГОСТ Р ИСО МЭК 9126-93 Информационная технология. Оценка программной продукции. Характеристики качества и руководства по их применению.
19. ГОСТ Р ИСО/МЭК 12119-2000 Информационная технология. Пакеты программ. Требования к уровню качества и тестирование.
20. *Нильсен Я.* Элементарные основы юзабилити. Электронный ресурс. [2018]. Режим доступа https://www.promo-webcom.by/analytics/usability/1429-elementarnyie-osnovyi-yuzabiliti/
21. *Седельников А.* Пять распространенных ошибок при разработке интерфейсов программ // Интернет-источник http://www.usability.ru/toader/mycolumn/five_errors.htm Центр практичных программ
22. *Соловьев С.В., Цой Р.И., Гринкруг Л.С.* Технология разработки прикладного программного обеспечения. Издательство: Академия Естествознания 2011 https://monographies.ru/en/book/view?id=141
23. *Сполский Дж.* Программистам о разработке пользовательских интерфейсов // Интернет-источник http://www.usability.ru/toader/articles/uid4p_1.htm Центр практичных программ
24. Человеко-машинный интерфейс. Правила организации http://genew.ru/cheloveko-mashinnij-interfejs-pravila-organizacii.html?page=6.
25. *Платт Д.* Софт-отстой и что с этим делать. Симбо, С. Петербург-Москва, 2008.
26. *Донской М.* Пользовательский интерфейс // Интернет-источник http://www.usability.ru/toader/articles/user_interface.htm Центр практичных программ
27. Интеллект человека и программы ЭВМ. Под ред. О.К. Тихомирова. М., 1979. С. 230.
28. *Лукин В.Н., Зотова А.А.* Пользовательский интерфейс: история и современность. “Новые информационные технологии”. Тезисы докладов XV Международной студенческой школы-семинара М.: МИЭМ, 2007. С. 50–55.
29. *Денинг В., Эссиг Г., Маас С.* Диалоговые системы “Человек-ЭВМ”. Адаптация к требованиям пользователя. М.: Мир, 1984. 112 с., научное издание.

30. *Раскин Дж.* “Интерфейс: новые направления в проектировании компьютерных систем”: Символ-Плюс; М.; 2005. 161 с., ил.

31. *Шнейдерман Б.* Психология программирования: Человеческие факторы в вычислительных и информационных системах. М.: Радио и связь, 1984. 303 с., рекомендация Минвуза.

32. Эмоциональный дизайн с примерами, перевод материала Smashing magazine, https://naikom.ru/blog/archives/6382

33. *Коутс Р., Влейминк И.* Интерфейс “Человек-Компьютер”. М.: Мир, 1990. 501 с., научное издание.

34. *Шнейдерман Бен.* Разработка пользовательского интерфейса: стратегии эффективного взаимодействия человека и компьютера, 1-е издание. Addison-Wesley, 1986; 2-е изд. 1992; 3-е изд. 1998; 4-е изд. 2005; 5-е изд. 2010; 6-е изд., 2016.

35. *Минанси М.* Графический интерфейс пользователя. Секреты проектирования. М.: Мир, 1996 г., 160 с., научное издание.

36. *Гультяев А., Машин И.* Проектирование и дизайн пользовательского интерфейса. СПб.: Корона принт, 2000. 352 с.

37. *Купер А., Рейман Р., Кронин Д.* Алан Купер об интерфейсе. Основы проектирования взаимодействия. Пер. с англ. СПб.: Символ Плюс, 2010. 688 с., ил.

38. *Лукин В.Н., Скобеева М.В., Чечиков Ю.Б. и др.* Автоматизированная биохимическая лаборатория. Перспективное развитие // Лабораторное дело. 1989. № 9. С. 26–28.

39. *Лукин В.В., Лукин В.Н., Лукин Т.В.* Технология разработки программного обеспечения. Уч. пособие, 4-е изд. М.: Вузовская книга, 2019. С. 294.

40. *Вятчин К.* Определение пользовательских профилей. Электронный ресурс. [2018]. Режим доступа: http://www.sdteam.com/t15690

41. *Латышев В.Л., Клыпина И.А.* Представление информации в системе “Человек-компьютер”. М.: Изд. МАИ, 1993. 22 с. Уч. пособие

42. *Кузнецов А.* Эмоциональный дизайн или тайна четвертой волны. Электронный ресурс. [2017]. Режим доступа: https://uexpert.ru/emotsionalnyj-dizajn-ili-tajna-chetvyortoj-volny/

43. *Латышев В.Л., Клыпина И.А.* Представление информации в системе “Человек-компьютер”. М.: Изд. МАИ, 1993. 22 с. Уч. пособие

44. *Уолтер А.* Эмоциональный веб-дизайн. Пер. с англ. Изд.: Манн, Иванов и Фербер, 2012. 93 с.: ил.

45. *Tognazzini B.* Вежливый интерфейс или принципы создания диалогов // Интернет-источник http://www.usability.ru/toader/articles.htm Центр практичных программ

46. *Нильсен Я.* Веб-дизайн. Книга Якоба Нильсена. Пер. с англ. СПб.: Символ-Плюс, 2003. 504 с.: ил.

47. *Эндрю вэн Дам* Пользовательские интерфейсы нового поколения // Открытые системы. 1997. № 6.

ПРОГРАММИРОВАНИЕ, 2020, № 5, с. 25–32

ПРОГРАММНАЯ ИНЖЕНЕРИЯ, ТЕСТИРОВАНИЕ И ВЕРИФИКАЦИЯ ПРОГРАММ

УДК 004.42

РАЗРАБОТКА КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ ЭНЕРГОСБЕРЕЖЕНИЯ УМНЫХ ЗДАНИЙ С ПРИМЕНЕНИЕМ КОМПЬЮТЕРНОЙ АЛГЕБРЫ

© 2020 г. Е. Ю. Щетинин*

Финансовый университет при правительстве РФ
125993 Москва, Ленинградский пр-т, д. 49, Россия
**E-mail: riviera-molto@mail.ru*

Поступила в редакцию 10.07.2019 г.
После доработки 15.09.2019 г.
Принята к публикации 03.10.2019 г.

Интеллектуальные технологии энергосбережения и энергоэффективности являются современным масштабным мировым трендом в развитии энергетических систем. Точные оценки энергосбережения важны для продвижения строительных проектов в области энергоэффективности и демонстрации их экономической привлекательности. Растущее количество цифровой измерительной инфраструктуры, используемой в коммерческих зданиях, привело к повышению доступности данных высокой частоты, которые можно использовать для обнаружения неисправностей и диагностики оборудования, отопления, вентиляции, и оптимизации кондиционирования воздуха. Это обусловило применение современных и эффективных методов машинного обучения, предоставляющих перспективные возможности для получения более точных прогнозов энергопотребления здания, и, таким образом, повышения показателей энергосбережения. В настоящей работе на основе модели градиентного бустинга предложен метод моделирования и прогнозирования энергопотребления комплекса зданий, а также разработаны компьютерные алгоритмы, их программно реализующие в системе компьютерной алгебры SymPy. Для оценки энергоэффективности были использованы данные о энергопотреблении 300 коммерческих зданий. Результаты компьютерного моделирования показали, что использование разработанных алгоритмов и программ повысило точность прогнозирования более чем в 80 процентах случаев по сравнению с другими алгоритмами машинного обучения.

DOI: 10.31857/S0132347420050088

1. ВВЕДЕНИЕ

Важнейшим из направлений развития экономики является повышение энергоэффективности производственных и потребительских секторов экономики. Одним из важнейших направлений развития экономики России утверждена государственная программа Российской Федерации “Энергосбережение и повышение энергетической эффективности на период до 2030 года”. В целях снижения воздействия на окружающую среду и затрат, связанных с сектором коммерческих зданий, было реализовано несколько программ по повышению энергоэффективности. Например, на государственном и федеральном уровнях в России были установлены долгосрочные целевые показатели энергосбережения, и эти целевые показатели должны быть достигнуты с помощью программ энергоэффективности. Анализ энергоэффективности имеет решающее значение для определения тарифов для владельцев зданий, плательщиков коммунальных тарифов и поставщиков услуг [1–3].

Развитие интеллектуальных сетей в производстве, финансах и услугах создает новые возможности для разработки и применения эффективных методов машинного обучения и анализа данных, проектирования новых модулей управления киберфизическими энергетическими системами. Внедрение интеллектуальных счетчиков обеспечивает преимущества конечным потребителям, поставщикам энергии и сетевым операторам, предоставляя потребителям информацию о режиме потребления, близком к реальному времени, что поможет им управлять реальным потреблением энергии, экономить деньги и сокращать выбросы парниковых газов. В то же время интеллектуальные счетчики способствуют планированию и эксплуатации распределительной сети, а также управлению спросом. В связи с этим данные интеллектуального учета позволят более точно прогнозировать спрос, повысить эффективность эксплуатации распределительных сетей, сократить время восстановления поставок, а также снизить эксплуатационные издержки сетей.

Интеллектуальные технологии сбора, регистрации и мониторинга данных о потреблении энергии создают огромное количество данных различного характера для использования поставщиками энергии и сетевыми операторами. Объем данных варьируется в зависимости от количества установленных интеллектуальных счетчиков, количества полученных сообщений интеллектуальных счетчиков, размера сообщения и частоты записи измерений, например, каждые 15 или 30 минут. Эти данные могут быть использованы для оптимального управления сетью, повышения точности прогнозирования нагрузки, выявления аномальных эффектов электроснабжения (условия пиковой нагрузки), формирования гибких ценовых тарифов для различных групп потребителей.

Основные модели, используемые в оценивании профилей энергопотребления, являются эмпирическими моделями, которые связывают объемы потребления электроэнергии в зданиях с такими параметрами, как, температура воздуха внешней среды, влажность, характеристики самого здания и т.д. Традиционно для построения таких моделей использовались данные ежемесячных счетов за коммунальные услуги, однако увеличение доступности данных счетчиков с часовым и 15-минутным интервалами позволило создать новые модели для более точных прогнозов. За последние десятилетия были достигнуты значительные успехи в разработке новых методов машинного обучения, среди которых наиболее перспективными с точки зрения точности прогнозирования являются подходы семейства алгоритмов ансамблевого обучения. Ансамблевые методы строят модель путем обучения нескольких относительно простых моделей, а затем объединяют их для создания более сложных моделей с более высокими прогностическими свойствами. В настоящей работе предложен новый компьютерный алгоритм построения модели потребления электроэнергии группой коммерческих зданий, использующих умные датчики регистрации показаний. Для этого нами были использованы алгоритмы машинного обучения на ансамблях, такие как случайный лес и градиентный бустинг GBM [4, 5, 13]. Также разработаны эффективные численные алгоритмы оценки и тонкой настройки параметров модели GBM и оценивания точности прогноза энергопотребления, которые были программно реализованы в системе компьютерной алгебры SymPy и языке программирования Python.

2. МЕТОДЫ МОДЕЛИРОВАНИЯ ЭНЕРГОПОТРЕБЛЕНИЯ С ИСПОЛЬЗОВАНИЕМ АЛГОРИТМОВ МАШИННОГО ОБУЧЕНИЯ

На сегодняшний день растущая доступность данных со смарт-счетчиков и в сочетании с интеллектуальным анализом данных позволяет оптимизировать процесс энергопотребления за счет повышения уровня автоматизации при сохранении и повышении точности прогнозирования [1–3]. Основными моделями, используемыми при оценке профилей энергопотребления, являются регрессионные модели, связывающие потребление энергии в зданиях с такими параметрами, как, температура внешней среды, влажность, индивидуальные характеристики здания и т.д. [3, 7]. Для построения таких моделей традиционно использовались данные ежемесячных счетов за коммунальные услуги, однако рост доступных данных со смарт-счетчиков с часовым и 30-минутным интервалом позволяют создать новые методы для более точного прогнозирования. Однако, с внедрением интеллектуальных технологий сбора, анализа и контроля данных энергопотребления достигнут значительный прогресс в разработке новых методов с применением алгоритмов машинного обучения, среди которых наиболее перспективными с точки зрения точности прогнозирования являются семейства алгоритмов ансамблей. Хорошо известны алгоритмы, применяемые в этих целях, такие как регрессионные деревья, случайный лес [4], бэггинг [5], а также бустинг [13]. Методы ансамблей создают модель, обучающую несколько относительно простых моделей, а затем объединяющую их для создания более сложной, но с лучшими прогнозирующими свойствами. Хотя эти алгоритмы машинного обучения с большим успехом используются во многих областях, они только начинают применяться к задачам моделирования энергосбережения. Например, в работах [8, 12, 15] авторы использовали случайный лес для прогнозирования почасового потребления энергии.

В данной работе предложен новый алгоритм выбора и тонкой настройки параметров модели энергопотребления зданиями, входящими в бизнес-комплекс, использующий алгоритм градиентного бустинга GBM, а также разработаны алгоритмы оценивания точности прогнозирования энергопотребления. Применение алгоритмов ансамбля и деревьев решений в качестве метода регрессии имеет несколько преимуществ, одним из которых является то, что правила разделения представляют собой интуитивно понятный и простой графический способ визуализации результатов. Кроме того, по своей структуре они могут одновременно обрабатывать числовые и категориальные входные параметры. Они устойчивы к выбросам и могут эффективно работать с пропусками данных в пространстве входных параметров. Иерархическая структура дерева решений автоматически моделирует взаимодействие между входными параметрами и выполняет эффективный отбор переменных, например, если входной параметр вообще не используется, то

прогноз не зависит от этого входного параметра. Наконец, алгоритмы деревьев решений просты в реализации и вычислительно эффективны с большими объемами данных [9, 10]. Алгоритм GBM был впервые предложен для задач классификации в работе [13]. Его основной принцип заключается в том, что несколько простых моделей, называемых “слабыми” моделями обучения, должны быть объединены в одну итерационную схему для выбора параметров с целью получения так называемой “сильной” модели обучения, то есть модели с лучшей точностью прогнозирования. Таким образом, алгоритм GBM добавляет новое дерево решений на каждой итерации так, что наилучшим образом уменьшает функцию потерь. Алгоритм продолжает работать, пока не будет достигнуто максимальное количество итераций или пока не будет достигнута указанная точность. Это означает, что на каждом новом шаге деревья решений, добавленные в модель на предыдущих шагах, фиксируются. Таким образом, модель может быть улучшена в тех ее частях, где она еще оценивает остатки недостаточно хорошо.

Алгоритм GBM более эффективен, если на каждой итерации вклад добавленного дерева решений учитывается с использованием некоторого гиперпараметра, описывающий одну из важных характеристик алгоритма, именно, скорость обучения модели v. Одна из проблем выбора гиперпараметра v состоит в том, что для достижения требуемой точности ϵ, в зависимости от значения v, необходимо соответствующее число итераций m. А именно, чем меньше значение v, тем большее число итераций необходимо выполнить. Таким образом, необходимо разработать процедуру оптимального выбора гиперпараметров v, m модели GBM при заданной точности ϵ. Еще одной проблемой является наличие автокорреляции, которая, как известно, вносит дополнительные искажения в оценки параметров модели [16]. Решением изложенных выше проблем на наш взгляд является применение рандомизации в процессе построения алгоритма градиентного бустинга GBM. На каждой итерации вместо всей выборки для оценки дерева решений применяется ее подвыборка, извлекаемая случайным образом. На практике, однако, редко можно выделить достаточное количество точек данных для точной оценки прогностической эффективности моделей без влияния на качество оценки. Когда количество наблюдений недостаточно, уменьшение размера обучающего набора может привести к существенному снижению точности [6, 9, 17]. Поэтому, для оценки влияния размера подвыборки на качество подгонки модели необходимо использовать подвыборки различных размеров. В этой ситуации для решения поставленных задач нами был разработан численный алгоритм выбора оптимальных значений гиперпараметров модели GBM с применением процедуры k-кратной перекрестной проверки и рандомизации (k-fold cross validation). Метод k-кратной перекрестной проверки включает деление набора данных на k подвыборок примерно одинакового размера. Сначала модель оценивается с использованием $(k-1)$ блоков в качестве обучающей выборки, а k-й блок (тестовая выборка) используется для определения точности прогноза. Далее процедура повторяется k раз, и всякий раз в качестве тестовой выборки используется новый блок. Для оценки точности прогноза на k-м блоке мы используем среднеквадратичную ошибку $RMSE_k = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y})^2}$, где $\hat{y}$ – прогнозируемое значение, y_i – значение набора реальных данных. Метод перекрестной проверки k-кратности использует среднеквадратичную ошибку в виде

$$RMSE(CV) = \frac{1}{k}\sum_{l=1}^{k} RMSE_l, \qquad (2.1)$$

k – число блоков. Предложенная и исследуемая в работе модель GBM имеет четыре гиперпараметра:

1. d – глубина деревьев решений;

2. m – количество итераций;

3. v – скорость обучения, которая обычно составляет значение между 0 и 1, уменьшение которого приводит к замедлению процедуры оценки, что требует увеличения количества итераций m;

4. k – количество разбиений выборки на подвыборки, используемые на каждом шаге итерации в алгоритме k-кратной перекрестной проверки (2.1).

Как и для любого алгоритма машинного обучения, проблема переобучения актуальна и для алгоритма GBM. Переобучение обычно является недостатком чрезмерно сложной модели. В случае модели GBM это может произойти, если выбрано слишком много итераций m или слишком большая глубина d деревьев решений. Таким образом, необходимо выбрать оптимальную комбинацию гиперпараметров, чтобы избежать переобучения и в то же время обеспечить наилучшую точность прогноза. Эффективным методом для решения этой проблемы является поиск по сетке (Grid Search) [14]. Этот подход состоит из определения множества различных значений гиперпараметров, построения модели для каждой их комбинации и выбора оптимальной комбинации с использованием показателей, которые количественно определяют модель с точки зрения точности прогнозирования (2.1).

Псевдокод представленного алгоритма имеет следующий вид:

1. Задать глубину деревьев решений d, количество итераций m, скорость обучения v, достижимую точность оценивания параметров модели ϵ.

2. Начальный шаг равен $x_0 = \bar{y}$, где $\bar{y}$ – среднее значение y, $f_0 = 0$.

3. Для $j = 1,\ldots,m$ выполнить:

• Произвольно извлечь подвыборку (x_i, y_i), $i = 1,\ldots,N$ из обучающей выборки, где N – количество измерений, соответствующих размеру подвыборки;

• Используя значения (x_i, y_i), построить дерево решений $\tilde{f}^j$ глубины d с учетом остатков z^j;

• Обновить значение остатков $z^{j+1} = z^j + v\tilde{f}^j(x)$.

4. Если $RMSE(CV) < \epsilon$ или $j = m$, то переход в п. 5. Если нет, то возврат к п. 3. (здесь ϵ – заданная точность оценки параметров модели).

5. Конец.

При реализации рассмотренного алгоритма перед нами возникла задача выбора системы компьютерной программы. Ряд содержательных рекомендаций по этому вопросу дан в работе [23]. Наиболее интересной на наш взгляд представляется система SymPy [24, 25]. Эта система появилась как библиотека символьных вычислений для языка Python [26]. Быстрое развитие этого языка в последние годы наряду с библиотеками Mathplotlib, SciPy, NumPy и др. обеспечило устойчивый интерес исследователей к применению в своей работе систем символьных вычислений. Удобный интерфейс SymPy позволяет передавать выходные результаты в библиотеки Python для их визуализации или дальнейших вычислений. В компьютерной среде языка Python широко представлены численные алгоритмы, реализующие метод *GridSearch* [26]. Для этого нужно импортировать класс *GridSearchCV* из библиотеки *sklearn.modelselection* и задать начальные значения основных параметров. Соответствующие операторы, программно реализующие выше изложенный алгоритм в Python, выглядят следующим образом

```
import numpy as np
from sympy import*
from sklearn.model_selection import
(cross_val_score, train_test_split,
GridSearchCV, RandomizedSearchCV)
from sklearn.metrics import r2_score
from lightgbm.sklearn import LGBMRe-
gressor
from sklearn.ensemble import
RandomForestRegressor
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.2,
random_state=2018)
```

Далее необходимо определить начальные значения гиперпараметров с использованием следующего оператора

```
hyper_space =
{'m_iter': [100, 200, 300, 400,…1000],
 'max_depth': [4, 5, 8],
 'num_leaves': [15, 31, 63, 127],
 'subsample': [0.6, 0.7, 0.8, 1.0]}
```

провести оценивание методом градиентного бустинга

```
est = LGBMRegressor(boosting='gbdt',
n_jobs=-1, random_state=2018)
```

и найти оптимальные значения гиперпараметров методом поиска по сетке *GridSearch*

```
gs = GridSearchCV(est, hyper_space,
scoring='r2', cv=4, verbose=1)
gs_results = gs.fit(X_train, y_train)
print("BEST PARAMETERS: "
+ str(gs_results.best_params_))
print("BEST CV SCORE: "
+ str(gs_results.best_score_)).
```

Далее построим прогноз энергосбережения и оценим его ошибку

```
# Predict (after fitting GridSearch-
CVis an estimator with best parameters)
y_pred = gs.predict(X_test)
# Score
score = r2_score(y_test, y_pred)
print("R2 SCORE: {}".format(score)).
```

При необходимости возможен вывод результатов работы программы в формат *LATEX*. Для этого следует импортировать модуль *LATEX*

```
from IPython.display import Latex
```

и вызвать функцию

```
sympy.init_printing(use_uni-
code=True}.
```

Кроме этого, возможно экспортировать результаты в формат ряда языков программирования (*C*, *C++*, *Fortran*). Данные функции доступны с использованием класса *sympy.printing*.

3. КОМПЬЮТЕРНЫЕ ЭКСПЕРИМЕНТЫ И АНАЛИЗ РЕЗУЛЬТАТОВ

Для тестирования разработанных в статье алгоритмов были использованы данные показателей электропотребления городского конгломерата зданий в г. Дублин, Ирландия [11]. Данные представляют собой 15-минутные измерения потребления электроэнергии в кВт за период 29.03.2011–20.02.2013. Типовой график исследуемых временных рядов энергопотребления одного из зданий приведен на рис. 1. Для каждого здания

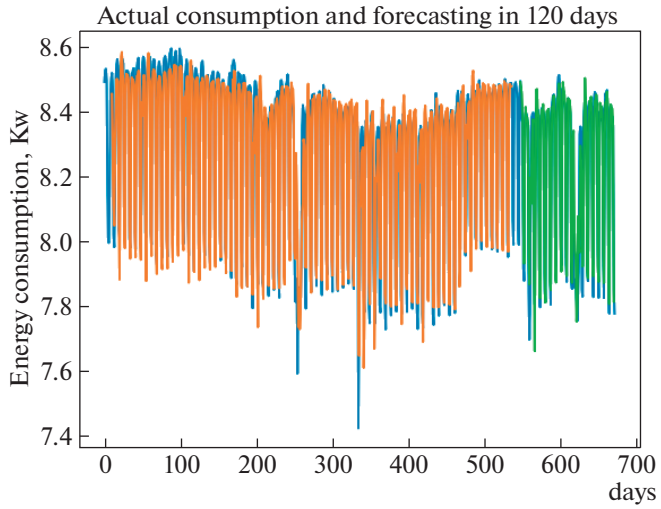

Рис. 1. График энергопотребления одного из зданий в логарифмической шкале.

данные разбиты на периоды обучения и прогнозирования. Периоды прогнозирования были определены в пределах последних 12 месяцев исследуемых данных. Модели регрессии, случайного леса и градиентного бустинга обучались с использованием двух разных периодов, которые составили 6 и 12 месяцев. Глубина деревьев решений d была выбрана из набора (3, ..., 10), скорость обучения v была выбрана между 0.05 и 1, число итераций m принимало значения от 1 до 1000 с шагом в 10 итераций. Гиперпараметры GBM настраивались с использованием алгоритма поиска по сетке совместно с методами перекрестной проверки. Для этого были использованы три варианта стандартной процедуры перекрестной проверки (CV): 5-кратная CV, 5-кратная CV и сутки в качестве блока (CV-1day), 5-кратная CV с одной неделей в качестве блока для CV (CV-7day). Таким образом, с учетом выше предложенных алгоритмов перекрестной проверки с тонкой настройкой параметров, имеем три различные реализации модели градиентного бустинга: 1) модель GBM с выбором параметров стандартным методом 5-кратной перекрестной проверки CV; 2) модель GBM с CV-1day (GBM-1d); 3) модель GBM с CV-7day (GBM-7d).

Результаты компьютерных экспериментов показали, что значение скорости обучения $v = 0.1$ слишком мало, так как алгоритм оказался слишком чувствителен как к количеству итераций, так и к глубине деревьев решений. Также выяснилось, что при скорости обучения 0.2 и глубине дерева решений $d = 5$ алгоритм не достиг оптимального числа итераций при $m = 500$. При увеличении скорости обучения до $v = 1$ из-за необходимости увеличения количества итераций и в силу возросшей сложности модели алгоритм начинает переобучаться. Таким образом, оптимальная скорость обучения была принята равной 0.5. Характер поведения ошибки RMSE (2.1) модели GBM в зависимости от различных значений v представлен на рис. 2.

Показатели точности коэффициент детерминации R^2, среднеквадратическая ошибка RMSE(CV) были рассчитаны для всего набора данных и продемонстрировали снижение точности при сокращении периода обучения с 12 до 6 месяцев. При этом компьютерные эксперименты показали, что R^2 для моделей GBM превосходит соответствующие значения R^2 для моделей RF и регрессии. Это осо-

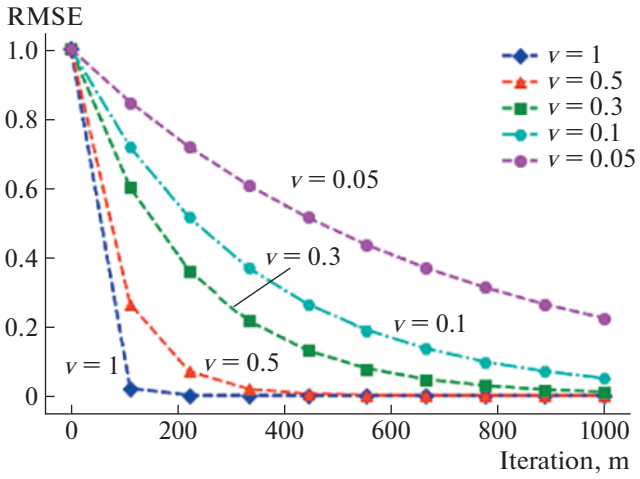

Рис. 2. RMSE для различных значений гиперпараметра v при обучении модели GBM.

Таблица 1. Оценки точности моделей энергопотребления на тестовой выборке для 6 и 12 месяцев

Модель	R^2 (6)	RMSE(CV) (6)	R^2 (12)	RMSE(CV) (12)
GBM	33	47	61	76
GBM - 1д	57	63	67	81
GBM - 7д	77	70	81	86
RF	28	40	35	48
Регрессия	17	30	27	38

бенно заметно для моделей GBM-1d и GBM-7d. Как и ожидалось, модель GBM, использующая стандартную процедуру CV, имеет меньшую точность, чем две другие версии, в которых используется k-кратная проверка. С уменьшением периода обучения до 6 месяцев в стандартной модели GBM наблюдалось некоторое снижение R^2 и его небольшое снижение для моделей GBM-1d, GBM-7d и RF. В то же время точность модели регрессии не улучшилась, что означает, что для этого набора данных регрессионная модель не повышает точность при увеличении количества наблюдений.

Что касается результатов оценки показателя точности RMSE(CV), то модели GBM-1 и GBM-7d превосходят модели RF и GBM в обоих периодах обучения. Точность прогноза моделей GBM значительно повысилась, когда их период обучения был увеличен с 6 до 12 месяцев, в то время как показатель точности регрессионной модели R^2 несколько снизился. Напомним, что для показателя R^2 желательны более высокие значения, тогда как для значений RMSE(CV) желательно иметь их близкими к нулю. В Таблице 1 показано количество зданий в процентах, для которых модели GBM оказались более точными, чем регрессионная и модель случайного леса RF. Для RMSE(CV) столбцы представляют процент зданий с меньшим значением RMSE(CV), чем у регрессионной модели и RF. Эти результаты подтверждают, что алгоритмы GBM-1d и GBM-7d имеют более высокую точность, чем регрессионные и RF модели. На основе построенных моделей также были проведены моделирование и прогноз энергопотребления на примере показателей одного из зданий. На рис. 1. представлен график прогноза на 120 дней с использованием алгоритма GBM-7d.

4. РЕЗУЛЬТАТЫ РАБОТЫ И ВЫВОДЫ

В настоящей работе предложен метод моделирования и оценивания потребления электроэнергии крупными коммерческими центрами и зданиями. Для оценивания параметров модели был использован алгоритм градиентного бустинга с адаптивной настройкой гиперпараметров с использованием k-кратной процедуры перекрестной проверки. Для вычисления оптимальных значений гиперпараметров был разработан компьютерный алгоритм с использованием поиска по сетке. Работоспособность и эффективность применения алгоритма GBM к решению задачи повышения точности прогноза потребления электроэнергии была протестирована на реальных данных электропотребления. Модель GBM показала более высокую точность прогнозирования, чем модели регрессии и случайного леса на всех протестированных периодах обучения. Результаты проведенных компьютерных экспериментов показали, что использование модели GBM позволяет повысить точность оценки энергосбережения как отдельного здания, так и комплекса зданий в целом. Также установлено, что использование 6-месячного периода обучения для построения моделей GBM привело к незначительному снижению точности прогноза энергопотребления, по сравнению с теми, которые были получены на 12-месячном периоде обучения, который обычно используется для всего здания. Подобное утверждение оказалось несправедливым для линейных моделей регрессионного анализа, а также ряда алгоритмов машинного обучения, что подтвердило результаты работ [7, 15]. Таким образом, применение алгоритмов GBM позволяет не только повысить точность оценки энергосбережения в целом, но и сократить общее время, необходимое для проведения оценки энергосбережения всего комплекса зданий.

Сравнительный анализ разработанных в статье алгоритмов настройки гиперпараметров показал, что важно принять во внимание автокорреляцию временных рядов. Действительно, результаты компьютерных экспериментов продемонстрировали, что использование стандартной процедуры перекрестной проверки снижает точность алгоритма GBM. Это связано с тем, что при использовании стандартного подхода к выбору гиперпараметров наблюдения в тестовых и обучающих выборках не являются независимыми (из-за автокорреляции измерений), что приводит к переобучению. С целью преодоления влияния автокорреляции в алгоритм оценки точности прогноза энергопотребления была включена процедура рандомизации. Также было показано, что различие в использовании в качестве периода прогноза одной недели или одного дня не оказывает существенного влияния на результаты оценивания и прогноза энергопотребления. Поэтому можно сделать вывод, что для большинства случаев использование одного дня в качестве стандартного блока при оце-

нивании точности прогноза энергопотребления является хорошим выбором.

Известно, что одним из главных достоинств модели ансамблей является их гибкость и надежность при использовании с большим количеством входных параметров [18]. Кроме того, по сравнению с регрессионными моделями, нет необходимости модифицировать алгоритм для обработки дополнительных входных параметров. Вместо этого достаточно включить эти переменные в качестве входных данных алгоритма без необходимости определять конкретную форму модели для каждого из параметров. Кроме того, дополнительные возможности выбора переменных в модели GBM позволяют исключать параметры, не влияющие на модель, без снижения свойств модели. Модель GBM обладает рядом очевидных преимуществ по сравнению с регрессионными моделями, состоящих в ее способности сохранять точность на более коротких периодах обучения, повышение общей точности показателей энергоэффективности и простота включения дополнительных объясняющих переменных. Такими параметрами могут служить, например, заполняемость здания, его освещённость и влажность. Дальнейшее развитие настоящей работы будет связано с применением модели GBM к решению проблем, связанных с энергоэффективностью [16, 17, 21], таких как прогнозирование потребления энергии при долго срочном планировании нагрузки, непрерывное обнаружение аномалий и количественная оценка изменения сетевой нагрузки, реагирующей на спрос [19, 20, 22].

Рассмотренные алгоритмы были программно реализованы в системе компьютерной алгебры SymPy и языка программирования Python. В работе приведены отдельные элементы программного комплекса и продемонстрированы результаты его работы для задач энергосбережения. Его применение значительно повысило эффективность исследований в указанной области. Можно сделать вывод, что использование систем компьютерной алгебры стало неотъемлимой частью в разработке компьютерных технологий в области энергоэффективности.

СПИСОКИ ЛИТЕРАТУРЫ

1. *Gellings C.W.* The smart grid: enabling energy efficiency and demand response. The Fairmont Press, Inc., 2009.
2. *Arghira N., Hawarah L., Ploix S., Jacomino M.* Prediction of appliances energy use in smart homes. Energy. 2012. V. 48. № 1. P. 128–134.
3. *Hawarah L., Jacomino M.* Smart Home – From User's Behavior to Prediction of Energy Consumption. ICINCO 2010, June 15–18, Proceedings of the 7th International Conference on Informatics in Control, Automation and Robotics, Volume 1, Funchal, Madeira, Portugal.
4. *Breiman L.* Random forests, Machine learning. 2001. V. 45. № 1. P. 5–32.
5. *Breiman L.* Bagging predictors, Machine learning. 1996. V. 24. № 2. P. 123–140.
6. *Bergmeir C., Benitez, J.M.* On the use of cross-validation for time series predictor evaluation. Information Sciences. 2012. V. 191. P. 192–213.
7. *Ediger V., Akar S.* ARIMA forecasting of primary energy demand by fuel in Turkey. Energy Policy. 2007. V. 35. P. 1701–1708.
8. *Cincotti S., Gallo G., Ponta L.* Modeling and forecasting of electricity spot-prices: Computational intelligence vs. classical econometrics. AI Commun. 2014. V. 27. P. 301–314.
9. *Ardakani F.J., Ardehali M.M.* Novel effects of demand side management data on accuracy of electrical energy consumption modeling and long-term forecasting. Energy Convers. Manag. 2014. V. 78. P. 745–752.
10. *Shchetinin E.Yu.* Cluster-based energy consumption forecasting in smart grids, Springer Communications in Computer and Information Science (CCIS). Springer, Berlin. 2018. V. 919. P. 446–456.
11. https://data.gov.ie/dataset/energy-consumption-gas-and-electricity-civic-offices-2009-2012/resource/
12. *Liaw A., Wiener M.* Classification and Regression by Random Forest. R News, 2002. V. 2. № 3. P. 18–22.
13. *Friedman J.* Greedy function approximation: a gradient boosting machine. Ann. Stat. 2001. V. 29. № 5. P. 1189–1232.
14. *Burnham K.P., Anderson D.R.* Model Selection and Multi-Model Inference: A Practical, Information-theoretic Approach, Springer Verlag, 2002.
15. *Kane M.J., Price N., Scotch M.* Comparison of ARIMA and Random Forest time series models for prediction of avian influenza H5N1 outbreaks, BMC Bioinformatics. 2014. V. 15. P. 276–284; https://doi.org/10.1186/1471-2105-15-276
16. *Granderson J., Touzani S., Custodio C.* Accuracy of automated measurement and verification techniques for energy savings in commercial buildings, Applied Energy. 2016. V. 173. P. 296–308.
17. *Jain R.K., Smith K.M., Culligan P.J., Taylor J.E.* Forecasting energy consumption of multi-family residential buildings using support vector regression: investigating the impact of temporal and spatial monitoring granularity on performance accuracy, Appl. Energy. 2014. V. 123. P. 168–178.
18. *Zhou Z.H.* Ensemble learning, Encycl. Biom. 2015. P. 411–416.
19. *Zorita A.L., Fernandez-Temprano M.A., Garcia-Escudero L.-A., Duque-Perez O.* A statistical modeling approach to detect anomalies in energetic efficiency of buildings, Energy Build. 2016. V. 110. P. 377–386.
20. *Amozegar M., Khorasani K.* An ensemble of dynamic neural network identifiers for fault detection and isolation of gas turbine engines, Neural Network. 2016. V. 76. P. 106–121.

21. *Shchetinin E.Yu., Melezhik V.S., Sevastyanov L.A.* Improving the energy efficiency of the smart buildings with the boosting algorithms, Proceedings of the Selected Papers of the 12th International Workshop on Applied Problems in Theory of Probabilities and Mathematical Statistics in the framework of the Conference on Information and Telecommunication Technologies and Mathematical Modeling of High-Tech Systems (APTP+MS'2018), CEUR Workshop Proceedings. 2018. V. 2332. P. 69–78.

22. *Lyubin P., Shchetinin E.Yu.* Fast two-dimensional smoothing with discrete cosine transform, Communications in Computer and Information Science (CCIS). Springer, Berlin. 2016. V. 678. P. 646–656.

23. *Геворкян М.Н., Демидова А.В., Велиева Т.Р., Королькова А.В., Кулябов Д.С., Севастьянов Л.А.* Реализация метода стохастизации одношаговых процессов в системе компьютерной алгебры. Программирование. 2018. № 2. С. 18–27.

24. *https://github.com/sympy*.

25. *https://www.sympy.org/ru/index.html*.

26. *https://www.python.org*.

ПРОГРАММИРОВАНИЕ, 2020, № 5, с. 33–46

ПРОГРАММНАЯ ИНЖЕНЕРИЯ, ТЕСТИРОВАНИЕ И ВЕРИФИКАЦИЯ ПРОГРАММ

УДК 004.421.6

NOBRAINER: ИНСТРУМЕНТ ПРЕОБРАЗОВАНИЯ C/C++ КОДА НА ОСНОВЕ ПРИМЕРОВ

© 2020 г. В. В. Савченко[a,*], К. С. Сорокин[a,], И. Е. Бронштейн[a,***], А. С. Волков[a,****], В. В. Качанов[a,*****], Г. А. Панкратенко[a,******], М. К. Ермаков[a,*******], С. И. Марков[a,********], А. В. Спиридонов[a,*********], И. В. Александров[a,**********]**

[a] *Институт системного программирования РАН им. В.П. Иванникова*
109004 Москва, ул. Александра Солженицына, д. 25, Россия

**E-mail: vsavchenko@ispras.ru*
***E-mail: ksorokin@ispras.ru*
****E-mail: ibronstein@ispras.ru*
*****E-mail: asvolkov@ispras.ru*
******E-mail: vkachanov@ispras.ru*
*******E-mail: gpankratenko@ispras.ru*
********E-mail: mermakov@ispras.ru*
*********E-mail: markov@ispras.ru*
**********E-mail: aspiridonov@ispras.ru*
***********E-mail: ialexandrov@ispras.ru*

Поступила в редакцию 10.02.2020 г.
После доработки 20.02.2020 г.
Принята к публикации 15.03.2020 г.

Рефакторинг – это неотъемлемая часть современного процесса разработки. Часто рефакторинг требуется выполнять на глобальном уровне с модификациями в большом количестве файлов. Внесение таких изменений вручную – долгая и кропотливая работа. Однако пользователи редко применяют автоматические инструменты для этих целей, так как считают их ненадежными и сложными в использовании.

В этой статье представлен новый инструмент для выполнения трансформаций исходного кода. Он основан на интуитивной схеме задания пользователем правил преобразования в виде коротких фрагментов кода на языке C или C++. Правила описывают то, как код должен выглядеть до и после трансформации. Мы убеждены, что благодаря отсутствию дополнительных абстракций (таких как предметно-ориентированные языки), указанный подход гораздо проще применить на практике.

Несмотря на использование примеров с исходным кодом, представленный инструмент оперирует на уровне абстрактного синтаксического дерева. Это позволяет ему лучше анализировать пользовательский код и проверять корректность трансформаций.

DOI: 10.31857/S0132347420040056

1. ВВЕДЕНИЕ

Любой программный продукт постоянно эволюционирует. Эволюция в данном случае – это не только появление нового кода при расширении функциональности, но и непрерывный процесс модификации существующего. Чрезмерное внимание к первому может привести к быстрому накоплению технического долга в рамках проекта. *Технический долг* [2] – это метафора программной инженерии, обозначающая упрощения в архитектуре и коде, позволяющие ускорить первоначальные разработку и развертывание программного продукта. С течением времени, если технический долг не “выплачивать”, он может обрастать “процентами” – дополнительным временем, которое разработчики потратят на изменение программы. В худшем случае накопленные проблемы могут привести к невозможности продолжить разработку.

Стандартным методом борьбы с этим является *рефакторинг* [1], [12] – изменение внутренней структуры программы, которое при этом не влияет на ее функциональность [3]. Он помогает избавиться от существующих архитектурных проблем и упростить сопровождение программы в будущем. По оценкам Мерфи-Хилла и др. [8], 41% времени программисты тратят на деятельность,

связанную с рефакторингом. В той же работе приведены статистические данные, которые показывают, что разработчики предпочитают править код вручную, а не использовать автоматические инструменты для трансформации программ несмотря на то, что риск допустить ошибку в первом случае выше. Другое исследование, проведенное на сайте StackOverflow [9], показало, что такие инструменты как правило ненадежны, сложны в использовании и требуют слишком большого количества дополнительных действий от самого пользователя.

Все это позволяет сформулировать минимальные требования к инструменту рефакторинга, который можно считать полезным: **он должен быть прост в использовании и обеспечивать корректность проводимых трансформаций с учетом имеющейся синтаксической и семантической информации**.

Для языков C/C++ существуют популярные инструменты рефакторинга, такие как `Proteus` [4] и `Eclipse C++ Tooling` [5]. С их помощью пользователь может задавать правила для трансформации кода, используя *предметно-ориентированные языки* (domain-specific languages, далее – DSL). На таких языках можно выразить и то, какой рефакторинг требуется, и синтаксическую/семантическую структуру целевого языка программирования.

Однако с применением DSL к C/C++ есть проблемы. Исследования показывают, что по сравнению с другими популярными языками программирования обучение языкам C/C++ труднее [7], при этом в C/C++ проще допустить ошибку [10]. Получается, что с учетом DSL сложность использования инструмента рефакторинга многократно возрастает.

Таким образом, определение простоты использования можно уточнить: **инструмент рефакторинга не должен требовать от пользователя дополнительных узкоспециализированных знаний помимо знания самого C/C++**.

В данной статье представлен инструмент `Nobrainer`, предназначенный для проведения автоматических преобразований исходного кода программ на языках C/C++. Он основан на инфраструктуре `Clang/LLVM`[1] и соответствует всем вышеописанным требованиям. Название `Nobrainer` отражает его ключевую идею: это инструмент, который позволяет пользователям легко создавать и применять собственные правила для трансформации кода.

Правила для `Nobrainer` пишутся на C/C++ без использования DSL. Они неотличимы от обычного кода из проекта, что позволяет освоить инструмент в короткие сроки.

Далее мы опишем базовые принципы Nobrainer, продемонстрируем основные архитектурные и технические решения, а также приведем примеры его использования на промышленных проектах.

2. ОБЗОР СУЩЕСТВУЮЩИХ РЕШЕНИЙ

Эта глава посвящена существующим подходам к трансформации кода и автоматическому рефакторингу. В рамках обзора мы будем разделять два ключевых аспекта: форму описания трансформаций кода и то, каким способом эти трансформации производятся.

В большинстве описываемых здесь инструментов для правил трансформации кода используется собственный синтаксис. Например, в работе Уоддингтона (Waddington) и др. [4] вводится новый язык YATL, тогда как Лагода (Lahoda) и др. [6] расширяют язык Java, чтобы упростить написание таких правил. Мы считаем, что всевозможные DSL могут только запутать пользователя из-за своей дополнительной сложности. Авторы `ClangMR` [14] предлагают другой подход. Их инструмент используют *сопоставители абстрактных синтаксических деревьев* (Clang AST Matchers, далее – САСД) [11]. Они нужны для описания участков программного кода, которые необходимо трансформировать. Пользователь также должен составить замену для этих участков в терминах узлов *абстрактного синтаксического дерева* (AST, далее – АСД). Авторы предполагают, что пользователи разбираются в синтаксических деревьях вообще и в их построении для кода на C/C++ в частности. Мы же считаем, что это обычно не так, поэтому для рядового пользователя применение `ClangMR` может быть затруднительным.

Вассерман (Wasserman) [13] представил инструмент `Refaster`, предназначенный для рефакторинга кода на языке Java и не требующий никаких DSL. Автор предлагает использовать целевой язык программирования для описания правил трансформации. Это позволяет внедрить эти правила в кодовую базу проекта, что, в свою очередь, упрощает проверку их синтаксиса и семантических ограничений. Каждое правило пишется в форме класса, в котором есть методы с одной из двух аннотаций: `@BeforeTemplate` или `@AfterTemplate`. Такой класс описывает трансформацию и должен содержать как минимум один метод с `@BeforeTemplate` и ровно один метод с `@AfterTemplate`. Инструмент интерпретирует подобное описание следующим образом: он использует код в методах с `@BeforeTemplate` для поиска и сопоставления, после чего заменяет найденный код на то, что написано в

[1] https://clang.llvm.org/

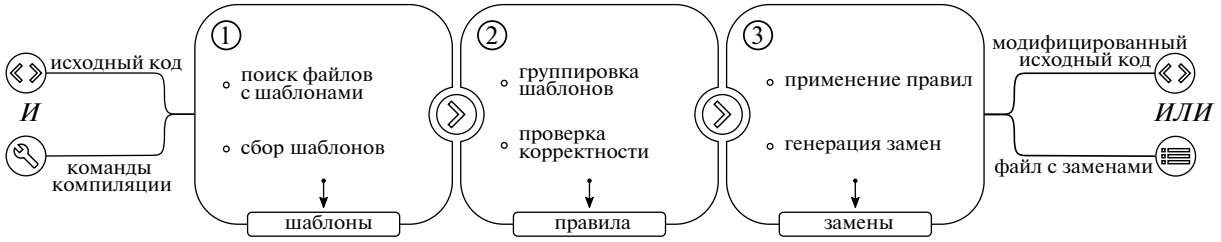

Рис. 1. Схема работы Nobrainer.

методе с `@AfterTemplate`. Поскольку указанный подход кажется нам наиболее понятным и удобным для конечных пользователей, он взят за основу нашего инструмента `Nobrainer`.

Мы также решили, что для поиска и сопоставления кода на C/C++ оптимальным является подход `ClangMR`. Однако использование САСД напрямую может быть затруднительным. По этой причине мы предоставляем более высокоуровневый фреймворк, который, в свою очередь, уже использует САСД.

В задаче трансформации кода стандартным решением является построение АСД, затем его преобразование и генерация кода. Такой подход используется инструментами `Proteus` [4], `Jackpot` [6] и `Eclipse C++ Tooling` [5]. Основной проблемой здесь является сохранение исходного форматирования на этапе генерации. Однако авторам `ClangMR` [14] удается избежать этой проблемы благодаря использованию для трансформации кода инфраструктуры `Clang/LLVM`. Это позволяет разработчикам напрямую править исходный код на уровне лексем. Мы также используем эту инфраструктуру, потому что полагаем, что это лучшее решение для трансформации программного кода на C/C++.

3. АРХИТЕКТУРА

В этой главе мы опишем общую архитектуру и схему работы `Nobrainer`.

Инструмент основан на использовании специальных примеров – фрагментов кода, написанных на языках C/C++. Каждый такой пример мы называем *шаблоном*, так как он может описывать целое семейство случаев. Сначала пользователь приводит пример языковых конструкций, которые он хотел бы заменить. Такой пример называется шаблоном `Before`. Затем в шаблоне `After` нужно написать код, который будет являться заменой для всех найденных конструкций. Описания шаблонов `Before` и `After` могут быть добавлены в любой, в том числе отдельный, файл пользовательского проекта.

Для запуска `Nobrainer` на каком-либо проекте необходимо сделать следующее:

• добавить в этот проект правила трансформации кода;

• указать команды компиляции для проекта (в настоящее время используются команды в формате `JSON`[2] из инфраструктуры `Clang/LLVM`).

На рис. 1 представлена внутренняя структура `Nobrainer`. Каждый пронумерованный блок соответствует одному этапу работы инструмента. Прямоугольные блоки в нижней части рисунка показывают, какие данные получаются на выходе каждого этапа.

Первый этап – поиск шаблонов `Before` и `After` в программном коде проекта.

Второй этап – составление списка правил трансформации путем группировки и проверки корректности шаблонов.

Третий этап – применение правил и генерация замен. Для каждого правила инструмент пытается сопоставить шаблоны `Before` с исходным кодом проекта. Для совпадений (успешных сопоставлений) по шаблону `After` формируются замены.

`Nobrainer` позволяет либо сразу применить к файлам проекта сформированные замены, либо сохранить их в формате `YAML`[3]. Во втором случае замены можно применить, используя специальный инструмент из набора `Clang Extra Tools`[4] – `clang-apply-replacements`.

Подробно все этапы описаны в главе 4.

4. ФОРМАЛЬНОЕ ОПИСАНИЕ И ДЕТАЛИ РЕАЛИЗАЦИИ

`Nobrainer` предоставляет программный интерфейс (API) для написания шаблонов. Он разделен на интерфейс для языка C и интерфейс для языка C++. Оба интерфейса дают возможность писать шаблоны для преобразования отдельных *выражений* (*expressions*) и последовательности

[2] https://clang.llvm.org/docs/JSONCompilationDatabase.html
[3] https://yaml.org/
[4] https://clang.llvm.org/extra/index.html

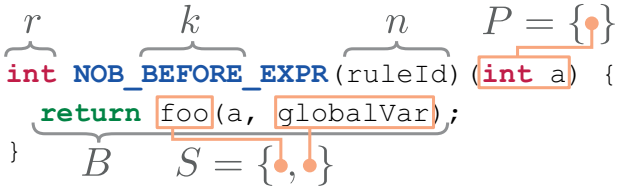

Рис. 2. Разбор шаблона Before.

операторов (*statements*) в исходных файлах на соответствующем языке.

Для более четкого понимания того, что из себя представляет шаблон, здесь и далее рассмотрим конкретный пример преобразования выражения. Предположим, что пользователь хочет найти все вызовы функции `foo` с двумя аргументами. Первый аргумент – это произвольное выражение типа `int`. Второй аргумент – это глобальная переменная `globalVar`. Указанные вызовы необходимо заменить вызовами функции `bar` с теми же аргументами. Листинг 1 содержит пример написания такого правила с использованием интерфейса для языка C.

```
int NOB_BEFORE_EXPR(ruleId)(int a) {
  return foo(a, globalVar);
}

int NOB_AFTER_EXPR(ruleId)(int a) {
  return bar(a, globalVar);
}
```

Листинг 1: Пример шаблона для преобразования выражения

Выражения для сопоставления и замены описываются внутри оператора `return`. Это сделано специально для того, чтобы делегировать проверку совместимости типов выражений из `Before` и `After` компилятору.

Трансформации инструмента `Nobrainer` основаны на том, что два выражения одинакового типа синтаксически являются взаимозаменяемыми. Это утверждение верно за исключением определенных ситуаций, когда выражения должны быть взяты в скобки. Существует набор специальных правил, которые позволяют избежать эту проблему, но в этой статье подробно они не рассматриваются.

Для того чтобы формально определить термин *шаблон* для данной программы, необходимо ввести следующую нотацию:

- Θ – конечное множество всех типов в данной программе;
- Σ – конечное множество всех определенных символов (функций, переменных, типов) в программе;
- $\mathscr{A}$ – конечное множество всех узлов АСД, из которых состоит данная программа;
- $\mathscr{C}$ – конечное множество символов, разрешенных для идентификаторов в языках C/C++;
- $\mathscr{P}$ – конечное множество всех параметров функции, $p \in \mathscr{P}$ и $p = \langle n_p, t_p \rangle$, где $n_p \in \mathscr{C}^*$ – это имя параметра и $t_p \in \Theta$ – это его тип.

Шаблон для замены выражения (expression template) можно формально описать как шестерку

$$T_{expr} = \langle k, n, r, B, P, S \rangle, \quad (1)$$

где

- $k \in \{\texttt{Before}, \texttt{After}\}$ – тип шаблона;
- $n \in \mathscr{C}^*$ – идентификатор правила, который используется для связывания шаблонов `Before/After`;
- $r \in \Theta$ – тип выражения;
- $B \subset \mathscr{A}$ – тело шаблона;
- $P \subseteq \mathscr{P}$ – множество параметров;
- $S \subseteq \Sigma$ – множество символов, которые используются в теле B.

Последние два элемента нуждаются в дополнительном пояснении.

Параметры шаблона P используются для выражения специальной семантики. `Nobrainer` рассматривает каждый параметр как произвольное выражение соответствующего типа. Например, параметр `a` в Листинге 1 соответствует **любому** выражению типа `int`.

Множество символов S используется для проверки корректности (см. Главы 4.2.3 и 4.3.2).

В следующих подглавах более подробно рассматриваются все стадии работы инструмента `Nobrainer`.

4.1. Поиск шаблонов

На первом этапе выполняется сбор и проверка корректности всех шаблонов, определенных в данном проекте.

4.1.1. Сбор шаблонов. `Nobrainer` обходит каждый файл и пытается найти функции, которые были объявлены с использованием программного интерфейса инструмента. Этот поиск можно выполнять только в синтаксически разобранных файлах. Если выполнять эту процедуру на всем проекте, это приведет к существенному замедлению работы инструмента. Этого можно избежать

путем обработки только тех файлов, которые содержат директивы `#include` с заголовочными файлами программного интерфейса `Nobrainer`.

В результате получается множество всех шаблонов, определенных пользователем. Обозначим это множество $\mathcal{T}$.

4.1.2. Проверка корректности шаблонов. Когда все шаблоны собраны, `Nobrainer` переходит к проверке корректности каждого отдельного шаблона. Здесь необходимо проверить, что шаблоны из $\mathcal{T}$ имеют правильную структуру. Важно отметить, что синтаксическая корректность шаблонов гарантируется компилятором. Их определения – это часть программного кода проекта, а значит для них проводится синтаксический разбор во время поиска. Это включает в себя проверку доступности всех используемых символов, проверку типов и др.

Каждый шаблон T_{expr} должен определять *ровно одно* выражение. Формально это правило может быть сформулировано следующим образом: тело шаблона B должно состоять из единственного непустого оператора `return`.

На текущий момент инструмент игнорирует шаблоны с использованием функциональных макросов (functional style macros) и лямбда-выражений языка C++. Это временное ограничение, которое мы планируем устранить в будущем.

Результатом работы данного этапа является множество корректных (с точки зрения вышеописанных правил) шаблонов, которое мы обозначим $\mathcal{T}_+$.

4.2. Составление списка правил преобразования

Для произвольного идентификатора $id \in \mathcal{C}^*$ определим две группы шаблонов B_{id} и A_{id} следующим образом:

$$B_{id} = \{T \in \mathcal{T}_+ | n_T = id, k_T = \texttt{Before}\} \quad (2)$$

$$A_{id} = \{T \in \mathcal{T}_+ | n_T = id, k_T = \texttt{After}\} \quad (3)$$

Две указанные группы описывают одну уникальную пользовательскую трансформацию кода, потому что они включают все шаблоны `Before` и `After` с одним и тем же именем. Однако для того чтобы B_{id} и A_{id} задавали правило преобразования, дополнительно должны выполняться следующие условия:

$$\begin{cases} |B_{id}| \geq 1 \\ A_{id} = \{a_{id}\} \text{ (т.е. } |A_{id}| = 1), \\ \forall b \in B_{id} \rightarrow a_{id} \prec b \end{cases} \quad (4)$$

где

$$\forall x, y \in \mathcal{T}_+ \rightarrow x \prec y \Leftrightarrow \begin{cases} P_x \subseteq P_y \\ r_x = r_y \end{cases} \quad (5)$$

Оператор $\prec$ – это *оператор совместимости*. Он задает отношение между шаблоном x и шаблоном y, которое указывает на то, что фрагмент кода, сопоставленный шаблону y, может быть безопасно заменен кодом шаблона x. Равенство возвращаемых типов r гарантирует тот факт, что заменяемое выражение имеет тот же тип, что и исходное. В то же время, условие $P_x \subseteq P_y$ позволяет быть уверенным в том, что во время замены инструмент будет иметь достаточно выражений для подстановки на место параметров из шаблона x.

Таким образом, можно определить *правило трансформации* как пару $R_{id} = \langle B_{id}, A_{id} \rangle$, где B_{id} и A_{id} удовлетворяют условиям (4). Множество всех правил, определенных в проекте, обозначим $\mathcal{R}$.

4.2.1. Обработка шаблонов `Before`. На этой стадии работы инструмента `Nobrainer` шаблоны Before преобразовываются в САСД. При помощи последних удобно искать поддеревья, удовлетворяющие заданным условиям. Каждый САСД состоит из предикатов для вершины поддерева и предикатов всех его узлов. Такая структура напоминает само синтаксическое дерево, и этот факт используется для генерации САСД в `Nobrainer`. Мы рекурсивно обходим все узлы дерева, построенного для тела шаблона `Before`, и для каждого типа узла генерируем свой САСД. Кроме того, для параметров шаблона создаются специальные САСД, которые будут связывать сопоставляемые поддеревья с именем параметра. В результате конструируется итоговое “дерево” из САСД. Таким образом, нам достаточно правильно реализовать генерацию САСД для каждого типа узла.

Рисунок 3 содержит упрощенный пример такого преобразования. На нем можно увидеть три представления шаблона `Before`: исходный код, синтаксическое дерево и САСД. Сплошные стрелки ведут от родительского к дочернему узлу АСД, а пунктирные показывают соответствие между узлом синтаксического дерева и САСД. САСД необходимо конструировать снизу вверх, поэтому АСД обходится в глубину.

4.2.2. Сопоставление одинаковых поддеревьев. Рассмотрим шаблон `Before` из Листинга 2. Маловероятно, что пользователь хочет найти вызов foo с двумя произвольными выражениями типа int в качестве аргументов. Наиболее интуитивная интерпретация такая: нужно найти вызов foo, в который передали два одинаковых выражения.

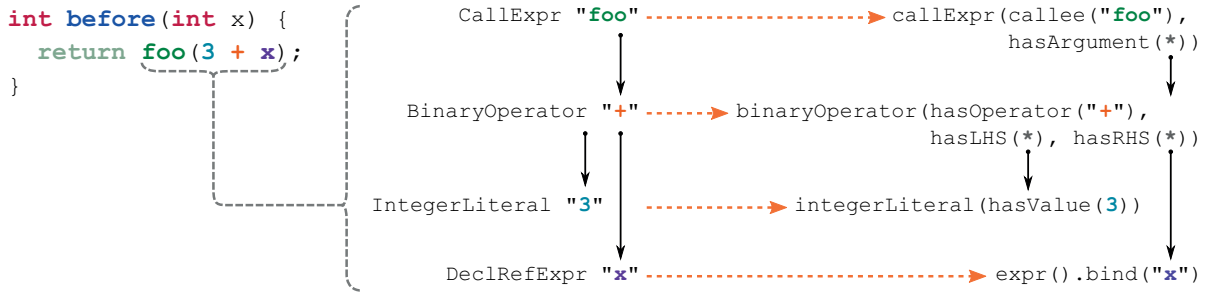

Рис. 3. Рекурсивная генерация САСД.

```
int before(int x) {
 return foo(x, x);
}
```

Листинг 2: Пример повторного использования шаблонного параметра

В инфраструктуре Clang/LLVM нет готового решения для сопоставления одинаковых поддеревьев. Используя уже имеющийся в Nobrainer механизм, мы динамически генерируем САСД во время процесса сопоставления. Это делается следующим образом: при сопоставлении первого аргумента вызова foo соответствующее поддерево связывается с параметром x. Затем по этому поддереву генерируется САСД, который и используется для сопоставления со вторым аргументом вызова.

4.2.3. Обработка шаблонов After. При обработке шаблона After наша цель – сконструировать текст, составляющий результат замены. Поэтому мы преобразовываем шаблоны After в форматные строки. Тело шаблона *B* может содержать элементы, у которых нельзя просто взять текстовое представление и использовать его в такой строке. Такие части мы называем *изменяемыми*. Во время обхода тела шаблона After мы извлекаем диапазоны, соответствующие изменяемым частям. Каждый диапазон содержит начальную и конечную позиции определенного узла синтаксического дерева. Есть два типа *изменяемых* частей.

Первый тип – это параметры шаблона в теле After, которые заполняются во время генерации замен (см. Подглаву 4.3.2).

Второй тип – символы (идентификаторы, не являющиеся параметрами шаблона). Вставка символов в произвольные места исходного кода может быть синтаксически некорректной, поскольку в том месте, куда символ вставляется, он может быть не объявлен. Поэтому мы собираем информацию о символах, которая затем используется во время генерации замен (см. Подглаву 4.3.2).

Например, для шаблона After из Листинга 3 сгенерируется следующая форматная строка: "${bar}($[x]) + 42". В этом примере Nobrainer выделяет символ bar и параметр x, помечая их соответствующим образом. Остальные части строки инструмент рассматривает как неизменяемые.

```
int after(int x) {
  return bar(x) + 42;
}
```

Листинг 3: Пример шаблона After

4.3. Применение правил и генерация замен

4.3.1. Применение правил. На следующем шаге нужно определить ситуации, в которых необходимо применять правила $\mathcal{R}$. Чтобы сделать это на всем проекте, Nobrainer проводит синтаксический разбор всех исходных файлов. После этого инструмент применяет САСД, сгенерированные для каждого правила.

Каждый раз, когда САСД находит какой-либо узел синтаксического дерева, удовлетворяющий всем предикатам, Nobrainer получает этот узел и список поддеревьев, связанных с параметрами соответствующего шаблона Before. Используя эту информацию и шаблон After, Nobrainer генерирует преобразование исходного кода, называемое *заменой*.

4.3.2. Генерация замен. Замена включает в себя:

- название файла, к которому применяется замена;
- смещение от начала файла для заменяемого текста;
- длину заменяемого текста;
- текст замены.

Первые три элемента Nobrainer получает из сопоставленного узла. Текст замены конструируется из форматной строки для шаблона After и поддеревьев, связанных с параметрами шаблона. Для каждого такого поддерева инструмент получает его текстовое представление из исходного

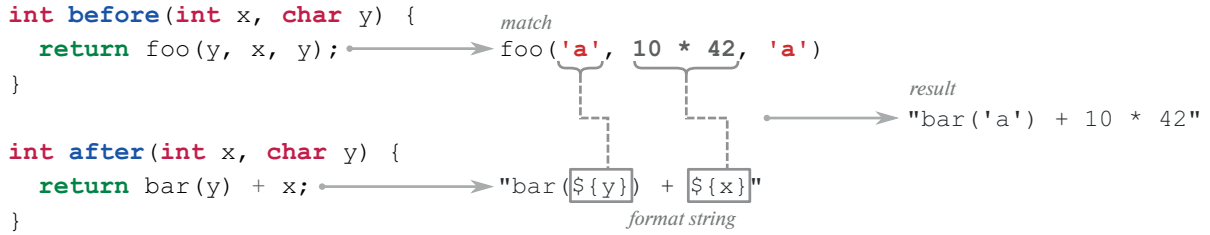

Рис. 4. Генерация замены.

кода, а затем подставляет эту подстроку в форматную строку вместо соответствующего параметра. На рис. 4 показана генерация текста замены для реального фрагмента кода.

Однако подобная замена может вызвать ошибки компиляции из-за недоступности каких-либо символов, появившихся в результате замены. `Nobrainer` должен обеспечить доступность этих символов. Для этого инструмент может добавлять:

• директивы `#include` для заголовочного файла, в котором определен данный символ;

• указание пространства имен или глобальной области видимости.

Таким образом, полученный фрагмент кода является корректным.

4.4. Типовые параметры

Использование произвольных выражений в качестве параметров шаблонов позволяет задавать правила в обобщенном виде. Однако, этого может быть недостаточно. Указание в правиле конкретных типов может сильно ограничить его выразительность и сузить круг применения.

Чтобы устранить этот недостаток, мы вводим в синтаксис шаблона множество типовых параметров $\Phi \subset \mathscr{C}^*$. Таким образом, мы расширяем определение шаблона для замены выражения (1) до следующего:

$$T'_{expr} = \langle k, n, r, B, P, S, \Phi \rangle, \tag{6}$$

а определение оператора совместимости $\prec$ (5) до:

$$\forall x, y \in T_+ \to x \prec y \Leftrightarrow \begin{cases} \Phi_x \subseteq \Phi_y \\ P_x \subseteq P_y \\ r_x = r_y \end{cases} \tag{7}$$

Заметим, что типовые параметры Φ полностью аналогичны параметрам P.

```
template <class T> T *before() {
  return (T *)malloc(sizeof(T));
}

template <class T> T *after() {
  return new T;
}
```

Листинг 4: Пример правила с типовым параметром

Листинг 4 демонстрирует правило, в котором используются типовые параметры.

4.5. Преобразование последовательности операторов

Инструмент `Nobrainer` может использоваться для трансформации не только отдельных выражений, но и последовательности операторов. Для обеспечения корректности таких преобразований мы расширяем определение шаблона T'_{expr} (6) до T_{stmt} следующим образом:

$$T_{stmt} = \langle k, n, r, P, B, S, \Phi, D \rangle, \tag{8}$$

где D – это конечное множество всех объявлений локальных переменных функции, $d \in D$ и $d = \langle n_d, t_d \rangle$, где $n_d \in \mathscr{C}^*$ – это имя параметра и $t_d \in \Theta$ – это его тип. Также мы усиливаем определение оператора совместимости $\prec$ (7) до:

$$\forall x, y \in \mathscr{T}_+ \to x \prec y \Leftrightarrow \begin{cases} \Phi_x \subseteq \Phi_y \\ P_x \subseteq P_y \\ D_x = D_y \\ r_x = r_y. \end{cases} \tag{9}$$

Таким образом, в шаблонах T_{stmt} все локальные переменные рассматриваются как параметры. Дополнительно мы требуем от пользователя не добавлять и не удалять объявления переменных.

Однако в определении (9) не учитываются области видимости локальных переменных, и инструмент не может гарантировать, что после замены код останется компилируемым. Это является временным ограничением. В Листинге 5 приведен пример, который при неаккуратном использовании может породить некомпилируемый код в том случае, когда счетчик используется вне цикла.

```
void before() {
  int i;
  for (i = 0; i < 10; ++i) {
    foo(i);
  }
}

void after() {
  for (int i = 0; i < 10; ++i) {
    foo(i);
 }
}
```

Листинг 5: Пример некорректного шаблона с изменением области видимости.

Также в шаблонах T_{stmt} допускается пустое тело функции After, что позволяет пользователю удалить некоторый код, сопоставленный с шаблонами Before.

4.6. Сопоставление произвольного оператора

С помощью трансформаций последовательности операторов можно изменять структуру кода в целом, поэтому было бы удобно обобщать правила преобразования. Для этого мы добавили специальную функцию `anystmt` в интерфейс инструмента `Nobrainer`. Ее вызов сопоставляется с любым оператором. Продемонстрируем применение `anystmt` на следующем примере (Листинг 6).

```
void before(nobrainer::Name x) {
  if (foo())
    anystmt(x);
}

void after(nobrainer::Name x) {
  if (!foo())
    x;
}
```

Листинг 6: Пример использования `anystmt`

Здесь мы хотим инвертировать условие в условном операторе с вызовом функции `foo`.

Как можно заметить, в списках аргументов шаблонов появились новые параметры типа `nobrainer::Name`. Это служебный тип, который мы ввели в интерфейс инструмента `Nobrainer`. В шаблоне `Before` параметр этого типа связывается с произвольным оператором, сопоставленным с вызовом `anystmt`. Это позволяет использовать связанный с соответствующим параметром оператор в шаблоне `After`.

Для сопоставления произвольного оператора в теле шаблона `Before` необходимо написать вызов функции anystmt с единственным параметром типа `nobrainer::Name`. Чтобы использовать сопоставленный с anystmt оператор в шаблоне `After`, нужно указать связанное с ним имя параметра (Листинг 6).

Для обеспечения этой функциональности мы ввели дополнительный САСД, который сопоставляет вызов функции `anystmt` с произвольным оператором и связывает с ним имя единственного параметра этого вызова. Стоит отметить, что в текущей реализации подстановка оператора, связанного с параметром типа `nobrainer::Name`, в шаблон `After` порождает предупреждение компилятора “Неиспользуемый результат выражения”. Обе проблемы мы устраним в скором времени.

4.7. Сопоставление произвольной последовательности операторов

Для расширения возможностей трансформации структуры кода мы вводим две дополнительные специальные функции: `block` и `block_greedy`. Их вызовы сопоставляются с произвольной, в том числе и пустой, последовательностью операторов и работают аналогично ленивой (.*?) и жадной (.*) квантификации в регулярных выражениях соответственно. Функции `block` и `block_greedy` принимают в качестве единственного аргумента параметр типа `nobrainer::Name` и связывают его имя с сопоставленной последовательностью операторов. Чтобы использовать эту последовательность в шаблоне `After` достаточно написать это имя аналогично механизму для `anystmt`.

Рассматриваемая функциональность обеспечивается специальным САСД. Внутренняя реализация этого САСД основана на алгоритме из книги Фридла (Friedl) [15], но его описание выходит за рамки данной статьи. На следующем примере мы продемонстрируем разницу в использовании ленивого (`block`) и жадного (`block_greedy`) интерфейсов (Листинги 7 и 8).

Стоит отметить, что на данный момент допустимо только одно использование каждого параметра типа `nobrainer::Name`, используе-

мого с `block` или `block_greedy` в теле шаблона `Before`.

```
void before(Name x) {
  block(x);
  foo();
}

void after(Name x) {
  x;
}

void example() {
            // Первое сопоставление:
            // <= block(x);
  foo();    // <= foo();

            // Второе сопоставление:
  bar();    // <= block(x):
  foo();    // <= foo();

            // Третье сопоставление:
  bar();    // <= block(x);
  foo();    // <= foo();
}
```

Листинг 7: Пример использования `block`

```
void before(Name x) {
  block_greedy(x);
  foo();
}

void after(Name x) {
  x;
}

void example() {
          // Единственное сопоставление:
          //
  foo(); // <=*
  bar(); // *
          // * block(x);
  foo(); // *
  bar(); // <=*
          //
  foo(); // <= foo();
}
```

Листинг 8: Пример использования `block_greedy`

В случае, когда `nobrainer::Name` связывается с пустой последовательностью операторов, во время создания замены в шаблон `After` вместо соответствующего параметра будет подставлена пустая строка.

4.8. Небезопасные преобразования

До сих пор мы старались обеспечить полную корректность трансформаций, что иногда сильно ограничивает возможности использования инструмента `Nobrainer`. В связи с этим мы вводим еще один вид шаблонов – T_{unsafe}. Его определение не отличается от шаблона T_{stmt} (8), но он имеет ослабленный оператор соответствия $\prec$:

$$\forall x, y \in \mathcal{T}_+ \to x \prec y \Leftrightarrow \begin{cases} \Phi_x \subseteq \Phi_y \\ P_x \subseteq P_y \\ r_x = r_y. \end{cases} . \qquad (10)$$

Мы больше не требуем совпадения множеств D_x и D_y. Это, в свою очередь, позволяет добавлять и удалять локальные переменные, а также менять их тип в шаблоне `After`. С ввдениеам небезопасных шаблонов стала возможна реализация, например, следующей замены (Листинг 9):

```
void before_unsafe(char *s) {
  for (int i = 0; i < strlen(s); ++i) {
    foo(s + i);
  }
}

void after_unsafe(char *s) {
  int len = strlen(s);
  for (int i = 0, i < len; ++i) {
    foo(s + i);
  }
}
```

Листинг 9: Пример использования небезопасного шаблона

Однако ответственность за ошибки компиляции, связанные с изменением локальных объявлений в результате замены, ложится на пользователя.

5. РЕЗУЛЬТАТЫ

В этой главе описывается подход к тестированию инструмента `Nobrainer`, рассматриваются примеры правил преобразования и анализируется производительность.

5.1. Тестирование

Тесты для инструмента можно разделить на две группы. Первая группа состоит из модульных и интеграционных тестов для каждого этапа из Главы 3. В основном они используются для проверки корректности автоматически сгенерированных САСД для шаблонов `Before` (Подглава 4.2.1) и форматных строк для шаблонов `After` (Подглава 4.2.3).

Вторая группа – это регрессионные тесты, которые включают в себя несколько проектов с открытым исходным кодом. Для каждого проекта вручную были созданы файлы с заранее заданными правилами преобразования. Система регрессионного тестирования запускает `Nobrainer`, измеряет время выполнения и проверяет, что все заданные преобразования корректно применились и что проект остался компилируемым.

5.2. Примеры правил

В этой подглаве представлены некоторые примеры правил преобразования, которые поддерживаются инструментом `Nobrainer`.

Первый пример (Листинг 10) описывает правило, с помощью которого изменяется порядок аргументов во всех вызовах метода `compose`. `Nobrainer` заменит каждый вызов метода `a.compose(x, y)` у произвольного объекта `a` класса `Agent` на `a.compose(y, x)`.

Этот пример демонстрирует, как автоматически можно поменять местами аргументы вызова.

```
int NOB_BEFORE_EXPR(ChangeOrder)(
    Agent a, char *x, char *y) {
  return a.compose(x, y);
}

int NOB_AFTER_EXPR(ChangeOrder)(
    Agent a, char *x, char *y) {
  return a.compose(y, x);
}
```

Листинг 10. Пример правила для изменения порядка аргументов

Второй пример (Листинг 11) показывает, как можно использовать `Nobrainer` для упрощения исходного кода.

```
using namespace nobrainer;

class EmptyRefactoring : public ExprTemplate {
public:
  bool beforeSize(const std::string x) {
    return x.size() == 0;
  }

 bool beforeLength(const std::string x) {
   return x.length() == 0;
  }

  bool after(const std::string x) {
   return x.empty();
  }
};
```

Листинг 11: Пример правила для проверки строки на пустоту

Напомним, что каждое правило может содержать несколько шаблонов `Before`, но только один шаблон `After`. Использование нескольких шаблонов `Before` в данном случае позволяет сгруппировать логически связанные трансформации.

Третий пример использует два вида параметров: обычные и типовые. Листинг 12 содержит исходный код правила преобразования.

```
using namespace nobrainer;

class CastRefactoring : public ExprTemplate {
public:
  template <class T>
  T *before(const T *x) {
    return (T *)x;
  }

  template <class T>
  T *after(const T *x) {
    return const_cast<T *>(x);
  }
};
```

Листинг 12: Пример правила для приведения к неконстантному типу

Указанное правило находит все операции приведения к неконстантному типу в стиле языка C и заменяет их эквивалентными конструкциями `const_cast`. В данном случае параметр `x` – это произвольное выражение, которое имеет тип “указатель на любой константный тип”. Указанное выражение должно замениться на выражение точно такого же типа, но без квалификатора `const`. `Nobrainer` учитывает все эти требования и корректно выполняет замены.

Четвертый пример описывает упрощение условия оператора `if` с помощью `anystmt` (Листинг 13).

```
using namespace nobrainer;

class SwapBranches : public StmtTemplate {
  void before(Name x, Name y, bool cond) {
    if (!cond)
      anystmt(x);
    else
      anystmt(y);
  }

  void after(Name x, Name y, bool cond) {
    if (cond)
      y;
    else
      x;
  }
};
```

Листинг 13: Пример правила с использованием `anystmt`

Данное правило удаляет отрицание в условии и меняет местами ветви оператора `if`.

В **пятом примере** реализуется небезопасная трансформация (Листинг 2). Здесь мы меняем тип переменной цикла.

```
using namespace nobrainer;

class ChangeType : public UnsafeStmtTemplate {
  void before(const char *s, Name x) {
    for (int i = 0; i < strlen(s); ++i) {
      block(x);
    }
}

  void after(const char *s, Name x) {
    for (size_t i = 0; i < strlen(s); ++i) {
       x;
    }
  }
};
```

Листинг 14: Пример правила с использованием небезопасного шаблона

5.3. Производительность

Измерения производительности проводились путем запуска системы регрессионного тестирования пять раз. Для этого использовалась рабочая станция на базе Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz с 64 Гб оперативной памяти и ОС Ubuntu 16.04 LTS.

Таблица 1 содержит результаты измерений. Для каждого проекта отдельно представлены его размер (тыс. строк кода), количество замен, которые применяет `Nobrainer`, и непосредственно время работы инструмента на данном проекте. Последнее состоит из времени работы двух фаз. Время каждой фазы измерялось отдельно. Первая фаза – это синтаксический разбор исходного кода проекта. Вторая фаза включает в себя всю основную логику работы инструмента `Nobrainer` (см. Главу 3).

Из табл. 1 видно, что затраченное время сильно варьируется в зависимости от проекта. В частности, это верно для времени работы как первой фазы, так и второй фазы. На рис. 5 дополнительно приведены процентные соотношения для времени синтаксического разбора на каждом проекте из системы регрессионного тестирования. Результаты показывают, что разбор в среднем занимает более 81% всего времени работы. Тот факт, что время выполнения оставшихся операций в среднем занимает менее 20%, гово-

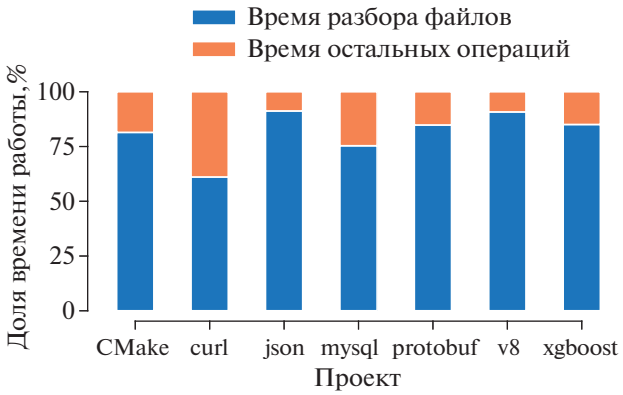

Рис. 5. Процент времени, затрачиваемого на синтаксический разбор.

Таблица 1. Производительность

Проект	Размер (тыс. строк)	Число замен	I фаза (сек)	II фаза (сек)
CMake	493	24	31.36	7.13
curl	129	7	3.17	2.01
json	70	7	13.99	1.34
mysql	1170	10	9.54	3.12
protobuf	264	8	16.62	2.97
v8	3055	6	281.57	28.52
xgboost	43	14	6.75	1.18

рит о том, что производительность инструмента Nobrainer близка к оптимальной.

6. ДАЛЬНЕЙШИЕ ИССЛЕДОВАНИЯ

На текущий момент Nobrainer поддерживает правила замены выражений и операторов. Кроме того, есть поддержка правил с типовыми параметрами. Инструмент уже можно использовать в системах непрерывной интеграции (CI), однако время работы все еще не позволяет применять его интерактивно на больших проектах. В целом, можно выделить три основных направления исследований для дальнейшей работы:

1. поддержку оставшихся узлов АСД, соответствующих выражениям и операторам C/C++;

2. улучшение производительности;

3. повышение удобства использования.

Для улучшения производительности предполагается исследовать возможные подходы к уменьшению времени, затрачиваемого на синтаксический разбор. Есть два возможных пути. Во-первых, это оптимизация фазы поиска совпадений за счет игнорирования файлов, не содержащих символов из шаблонов Before. Во-вторых, это исследование возможности автоматической генерации предкомпилированных заголовочных файлов (PCH). Эти файлы должны уменьшить время синтаксического разбора заголовочных файлов анализируемого проекта.

Для повышения удобства существует несколько направлений. Сейчас Nobrainer выполняет трансформации сразу на всех файлах проекта. Необходимо предоставить пользователю возможность указывать части проекта, для которых следует выполнять трансформации. Также важно рассмотреть варианты интеграции с другими инструментами для разработчиков. Например, упростить взаимодействие с пользователем путем создания расширений для интерактивных сред разработки (IDE). Другой возможный вариант – это использование инструмента Nobrainer в качестве вспомогательного программного средства. Например, с его помощью можно пытаться автоматически исправлять ошибки и дефекты, найденные статическим анализатором.

7. ЗАКЛЮЧЕНИЕ

В данной статье мы представили Nobrainer – инструмент для выполнения автоматических трансформаций программ на языках C/C++. Он основан на двух принципах: простоте использования и корректности правил преобразования. В работе мы уделили большое внимание описанию модели, архитектуры и деталей реализации с обоснованием принятых решений, результатами и примерами.

Наши результаты показывают, что Nobrainer уже может использоваться в системах непрерывной интеграции для проведения трансформаций на промышленных проектах. Мы описали текущие недостатки и обозначили возможные направления для улучшений. В будущем мы планируем повысить удобство использования нашего инструмента, а также интегрировать его с другими программными средствами для разработчиков.

СПИСОК ЛИТЕРАТУРЫ

1. *Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, Raghvinder Sangwan, Carolyn Seaman, Kevin Sullivan, and Nico Zazworka*. Managing Technical Debt in Software-reliant Systems. In Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER '10, pages 47–52, New York, NY, USA, 2010. ACM.

2. *Ward Cunningham*. The WyCash Portfolio Management System. SIGPLAN OOPS Mess. December 1992. V. 4 (2). P. 29–30.

3. *Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts*. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, June 1999.

4. *Daniel G. Waddington and Bin Yao*. High-Fidelity C/C++ Code Transformation. Electronic Notes in Theoretical Computer Science. 01 2007. V. 141. P. 35–56.

5. *Emanuel Graf, Guido Zgraggen, and Peter Sommerlad*. Refactoring support for the C++ development tooling. In OOPSLA Companion, 2007.

6. *Jan Lahoda, Jan Bečička, and Ralph Benjamin Ruijs*. Custom Declarative Refactoring in NetBeans: Tool Demonstration. In Proceedings of the Fifth Workshop on Refactoring Tools, WRT'12, pages 63–64, New York, NY, USA, 2012. ACM.

7. *Leo A. Meyerovich and Ariel S. Rabkin*. Empirical Analysis of Programming Language Adoption. SIGPLAN Not., October 2013. V. 48 (10). P. 1–18.

8. *Emerson R. Murphy-Hill, Chris Parnin, and Andrew P. Black*. How we refactor, and how we know it. In ICSE, pages 287–297. IEEE, 2009.

9. *Gustavo H. Pinto and Fernando Kamei*. What Programmers Say About Refactoring Tools?: An Empirical Investigation of Stack Overflow. In Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools, WRT '13, pages 33–36, New York, NY, USA, 2013. ACM.

10. *Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov*. A Large-scale Study of Programming Languages and Code Quality in GitHub. Commun. ACM, September 2017. V. 60 (10). P. 91–100.

11. The Clang Team. Clang documentation: Matching the Clang AST. https://clang.llvm.org/docs/LibASTMatchers.html.

12. *Will Tracz*. Refactoring for Software Design Smells: Managing Technical Debt by Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. ACM SIGSOFT Software Engineering Notes. 2015. V. 40 (6). P. 36.

13. *Louis Wasserman*. Scalable, example-based refactorings with Refaster. In Proceedings of the 2013 ACM workshop on Workshop on refactoring tools, pages 25–28. ACM, 2013.

14. *Hyrum Wright, Daniel Jasper, Manuel Klimek, Chandler Carruth, and Zhanyong Wan*. Large-Scale Automated Refactoring Using ClangMR. In Proceedings of the 29th International Conference on Software Maintenance, 2013.

15. *Джеффри Фридл*. Регулярные выражения. Символ-Плюс. СПб., 2008.

ПРОГРАММИРОВАНИЕ, 2020, № 5, с. 47–59

ЯЗЫКИ, КОМПИЛЯТОРЫ И СИСТЕМЫ ПРОГРАММИРОВАНИЯ

УДК 519.6

БИБЛИОТЕКА ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ ДЛЯ ЯЗЫКА C++

© 2020 г. М. М. Краснов[a,*]

[a] *Институт прикладной математики им. М.В. Келдыша РАН*
125047 Москва, Миусская пл., 4, Россия

******E-mail: kmm@kiam.ru*

Поступила в редакцию 26.07.2019 г.
После доработки 30.07.2019 г.
Принята к публикации 08.11.2019 г.

Современные функциональные языки программирования, такие, как Haskell, Scala, ML, F# обладают свойствами, позволяющими с их помощью относительно легко реализовывать логически сложные алгоритмы. К таким свойствам можно отнести композицию функций, каррирование, метафункции (функции над функциями) и ряд других. Это позволяет путем комбинирования простых функций получать более сложные функции. В качестве примера сложного алгоритма упомянем разбор строки (парсер). В языке Haskell имеется библиотека Parsec, представляющая из себя набор элементарных парсеров, комбинируя которые, можно создать более сложные парсеры. Это делает ее относительно простой и в то же время мощной. В языке C++ большинство из этих возможностей в явном виде отсутствуют. В то же время язык C++ достаточно мощный и позволяет реализовать многие из свойств функциональных языков. В предлагаемой библиотеке делается попытка по мере возможности решить данную задачу.

DOI: 10.31857/S0132347420050040

1. ВВЕДЕНИЕ

Решение написать данную библиотеку функционального программирования (funcprog) было принято под впечатлением от языка Haskell [1] в процессе его изучения. Его мощная система типов, функциональная “чистота”, “ленивость”, бесконечные списки, поддержка каррирования функций и их композиции, монады и трансформеры монад, а также многое другое, делают этот язык мощным инструментом для написания логически сложных алгоритмов. Другие современные языки функционального программирования (Scala [2], ML [3], F# [4]) также обладают аналогичными свойствами. При изучении языка Haskell ставилась задача понять, как на нем можно было бы максимально легко реализовать парсер математических формул (калькулятор). При этом выяснилось, что парсер общего назначения в библиотеке языка уже есть, и реализовать на нем парсер формул оказалось несложной задачей.

В то же время основным языком, которым пользуются многие математики, в том числе и автор данной статьи, для решения повседневных задач (в основном для реализации численных методов газовой динамики), является язык C++. При решении таких задач логически сложные алгоритмы тоже часто встречаются (например, тот же разбор формул), и использовать возможности функциональных языков программирования для их реализации кажется весьма привлекательным. Но совместить в одной программе два языка представляется сложной задачей, поэтому встал вопрос – а можно ли, оставаясь в рамках языка C++, писать на нем в стиле языка Haskell, используя многие из его свойств, включая каррирование функций и их композицию, монады, “ленивые” вычисления и т.д. Речь, по сути, идет о реализации в виде библиотеки классов и функций предметно-ориентированного языка (DSL, Domain Specific Language) функционального программирования.

Основное свойство всех функциональных языков – то, что функция в них является полноправным участником вычислений, т.е. функцию можно передать в качестве параметра функции и вернуть как результат вычислений. В частности, во всех функциональных языках есть понятие лямбда-выражения, результатом которого является функция, которую можно передать как параметр в другую функцию или вернуть как результат работы. В современном языке C++ такое понятие также есть, но появилось оно только в стандарте языка от 2011 года. Так как без поддерживаемых языком лямбда-выражений реализовать библио-

теку функционального программирования представляется сложным, да и не нужным (компилятор с поддержкой стандарта 2011 года есть почти на всех системах), то было принято решение не поддерживать более ранние версии языка. Библиотек функционального программирования для языка C++ много (см., например, [5, 6]). Особенностью данной библиотеки является, во-первых, то, что в ней делается попытка максимально полно реализовать функционал языка Haskell и, во-вторых, максимально используются возможности современного стандарта языка C++ от 2011 года и более поздних. При реализации библиотеки активно используется метапрограммирование шаблонов языка C++ (см. [7–9]). Далее дается описание модулей, реализованных в рамках библиотеки funcprog.

2. БАЗОВЫЕ КОМПОНЕНТЫ БИБЛИОТЕКИ

В данной библиотеке максимально используются имеющиеся возможности, во-первых, стандартной библиотеки языка C++, и, во-вторых, библиотеки boost [10]. Библиотека boost является полигоном для развития языка C++, прежде всего – его стандартной библиотеки. Многое из boost-а уже перекочевало в нее, что-то ждет своего часа. В частности, под функцией в библиотеке funcprog подразумевается объект класса *std::function* (в библиотеке boost это класс *boost::function*). Объект этого класса может быть обвязкой обычной функции, лямбда-функции (функции, порожденной лямбда-выражением) или функционального объекта (объекта, имеющего функциональный оператор). Таким образом, если функция принимает аргумент-функцию, то это значит (в рамках данной библиотеки), что она принимает аргумент класса *std::function*, а если функция возвращает функцию, то это значит, что она возвращает объект класса *std::function*. Шаблон класса *std::function* имеет один аргумент вида *Ret*(*Args*), где *Ret* – тип возвращаемого значения, а *Args* – список типов аргументов. Например, функция, принимающая аргументы типа *int* и *double* и возвращающая строку, имеет следующее описание:

```
std::function<std::string (int, double )>f;
```

Также в библиотеке активно используется переменное число параметров шаблона функции (variadic templates). Например, функция, принимающая первый параметр типа int и, возможно, еще какие-то параметры и возвращающая результат типа double, может быть описана так:

```
template<typename ... Types>
std::function<double (int , Types ... )> f;
```

Для реализации класса *Maybe* (который может содержать или нет значение некоторого типа) используется класс *boost::optional*, а если используется C++ стандарта 2017 г., то *std::optional*, а для класса *Either* (который содержит значение одного из двух типов) используется класс *boost::variant*, а если используется C++ стандарта 2017 г., то *std::variant*.

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

Как уже писалось выше, при написании библиотеки ставилась задача как можно полнее реализовать функционал языка Haskell. Но языки C++ и Haskell отличаются очень сильно, и поэтому может возникнуть законный вопрос: в какой степени возможности языка Haskell реализуемы на C++? Неполный список нетривиальных для реализации вещей включает ленивые вычисления, каррирование функций, η (эта)-редукцию (или η-преобразование), бесконечные списки, мощную систему вывода типов, конструкторы типов, полиморфные функции и много другое. Кое-что из перечисленного реализовать не удалось (например, бесконечные списки и η-редукцию), что-то – с определенными ограничениями (полиморфные функции), что-то реализовать удалось достаточно полно, но выглядит это менее элегантно, чем в Haskell-е. Особенно следует упомянуть переопределенные операторы. В языке Haskell почти любую последовательность символов можно определить как оператор, и в языке и его библиотеке это активно используется. С другой стороны, в языке C++ мы ограничены только тем набором операторов, которые изначально встроены в язык. Мы их можем переопределять для наших типов, но добавить новый оператор язык не позволяет. При реализации библиотеки те операторы языка Haskell, для которых есть аналоги в C++, сохранялись, другим (наиболее популярным) подыскивалась замена из числа имеющихся, часть операторов или заменялась на функции, или просто оставалась без реализации.

3.1. Функции

Как уже говорилось выше, под функцией в библиотеке подразумевается объект класса *std::function* (по умолчанию) или *boost::function*. Конкретный класс, реализующий функцию, скрыт за типом *function_t*:

```
template<typename F>
using function_t = std::function<F>;
```

В дальнейшем, чтобы избегать неоднозначности в термине “функция”, словом “функция” будем обозначать объект класса *function_t*, а про обычную функцию так всегда и будем писать – “обычная функция”. Любая обычная функция или функциональный объект (в том числе результат лямбда-выражения) могут быть преобразованы в функцию (к классу *function_t*). Для автоматического такого преобразования в библиотеке имеется специальная обычная функция с односимвольным именем _ (подчерк):

```
template<typename F>
function_type_t<F> _(F f) { return f; }
```

Эта обычная функция принимает в качестве параметра или указатель на обычную функцию, или ссылку на функциональный объект (в т.ч. на результат лямбда-выражения). Для преобразования типа параметра в тип функции используется метафункция *function_type*:

```
template<typename Ret , typename ... Args>
struct function_type {
using type = function_t <Ret (Args ... ) >;};

// Common case for functors & lambdas
template<class F> struct function_type:
 function_type<decltype (&F : : operator (
) ) >{};

template<typename Ret , class Cls ,
 typename ... Args>
struct function_type
<Ret ( Cls : : * ) (Args ... )> :
 function_type_<Ret , Args ... > { } ;

template<typename Ret , class Cls ,
 typename ... Args>
struct function_type
<Ret ( Cls : : * ) (Args ... ) const> :
 function_type_<Ret , Args ... > { } ;

// Common functions
template<typename Ret , typename ... Args>
struct function_type <Ret ( * ) (Args ... )> :
 function_type_<Ret , Args ... > { } ;

template<typename F>
using function_type_t =
   typename function_type <F>: : type ;
```

3.2. Ленивые вычисления

Допустим, нам надо описать прототип некоторой функции *f*, принимающей аргумент некоторого типа *a* и возвращающей результат некоторого типа *b*. В языке Haskell прототип такой функции записывается так: f::a–>b. При этом фактически в качестве параметра этой функции можно передать или значение (константу) типа *a*, или функцию без параметров, возвращающую результат типа *a*. С точки зрения языка Haskell это одно и то же. В языке C++ этому прототипу можно соотнести три разных прототипа:

```
b f ( a ) ;
b f ( a const&) ;
b f ( function_t<a ( )> const&) ;
```

Первый из этих прототипов принимает параметр по значению, второй – по константной ссылке и третий – функцию без параметров. С точки зрения языка C++ это три разных прототипа. Как же на C++ написать функцию, “готовую” к ленивым вычислениям, т.е. функцию, которой в качестве параметра можно было бы передать как значение нужного типа, как и функцию без параметров, возвращающую значение этого типа? В библиотеке funcprog для этого определяются несколько типов и функций. Во-первых, есть специальный тип для функций без параметров:

```
template<typename T>
using f0 = function_t<T( ) >;
```

Далее, есть функция *value_of*, которая, если ей в качестве параметра передать ссылку на функцию без параметров, вызывает ее и возвращает результат этой функции, в остальных случаях (если параметр не является ссылкой на функцию без параметров), просто возвращает этот параметр (его копию):

```
template<typename T>
T value_of (T const& arg )
{ return arg ; }

template<typename T>
T value_of ( f0<T> const& f )
{ return f ( ) ; }
```

Имеется метафункция *remove_f0*, преобразующая тип функции без параметров в простой тип:

```
template<typename T>
struct remove_f0
{ typedef T type ; } ;

template<typename T>
struct remove_f0<f0<T> >
{ typedef T type ; } ;

template<typename T>
using remove_f0_t =
  typename remove_f0<T>: : type ;
```

Теперь мы можем написать функцию, принимающую как значение некоторого типа, так и функцию без параметров:

```
template<typename Arg>
remove_f0_t<Arg> myfunc (Arg const& a ) {
  const remove_f0_t<Arg> a_ =
    value_of ( a ) ;
  return a_ * a_;
}
```

Обратиться к этой функции можно, например, так:

```
const f0<int> f = [ ] ( ) { return 5 ; } ;
s td : : cout
 << myfunc ( 5 ) << std : : endl
 << myfunc ( f ) << std : : endl ;
```

3.3. Каррирование

Каррирование не является обязательным атрибутом функциональных языков программирования (в отличие, например, от лямбда-выражений, которые есть во всех языках функционального программирования). Например, в одном из первых языков функционального программирования Lisp каррирования нет. Тем не менее, этот механизм очень удобный и современные функциональные языки его, как привило, имеют. Смысл каррирования в следующем. Если мы при вызове функции укажем не все параметры (опустив несколько последних), то результатом такого вызова будет функция с меньшим числом параметров (равным числу опущенных параметров), при вызове которой будет вызвана исходная функция. На самом деле язык Haskell идет еще дальше. Даже если указать все параметры функции, то и в этом случае фактического вызова функции не происходит. Вместо этого создается функция без параметров, т.е. каррирование идет “до конца”. На этом основаны “ленивые” вычисления. Фактический вызов функции откладывается как можно дальше.

Каррирование в библиотеке funcprog реализовано с помощью перегруженных (overloaded) функций. Пусть имеется некоторая функция (например, с именем myfunc) с тремя параметрами. Тогда при определении этой функции будет создано дополнительно еще 2 перегруженных функции с именем _myfunc, имеющих один и два параметра и возвращающих, соответственно, функции, имеющие два и один параметр. Каждая из этих возвращаемых функций при вызове будет обращаться к функции myfunc. Вот как, например, выглядит определение функции liftA2:

```
DEFINE_FUNCTION_3( 3 ,LIFTA2_T(T0 ,T1 ,T2) ,
    l i f t A 2  , f u n c t i o n _ t <T2> const&, f ,
    T1 const&, x , T0 const&, y ,
    return f / x * y ; )
```

Макрос DEFINE_FUNCTION_3 создаст определение функции liftA2:

```
template<typename T0 , typename T1 ,
typename T2> LIFTA2_T(T0 , T1 , T2)
l i f t A 2 ( f u n c t i o n _ t <T2> const& f ,
  T1 const& x , T0 const& y )
{ return f / x * y ; }
```

а также определение двух перегруженных функций (с одним первым и с двумя первыми параметрами) с именем _liftA2. Вот как, например, будет выглядеть определение функции с двумя первыми параметрами:

```
template<typename T0 , typename T1 ,
  typename T2> f u n c t i o n _ t <
  LIFTA2_T(T0 ,T1 ,T2) (T0 const&)
> _ l i f t A 2 ( f u n c t i o n _ t <T2> const& f ,
    T1 const& x ) {
  return [ f , x ] (T0 const& y ) {
    return l i f t A 2 ( f , x , y ) ; } ;
}
```

3.4. Применение функции

Применение функции к значению для функций с одним параметром в языке C++ – это, по сути, просто вызов функции и, как известно, делается с помощью выражения типа f(x). В языке Haskell скобки опускаются и это выглядит как f x,

или, с использованием оператора $ (специального оператора применения функции к параметру), как f $ x. Кроме того, как уже говорилось выше, в языке Haskell фактического вызова функции при этом не происходит, вместо этого порождается функция без параметров.

В случае же функции с несколькими параметрами ситуация иная. В языке C++ мы не можем указать только часть параметров и при вызове функции должны всегда указывать все параметры, так что применение функции с несколькими параметрами к одному параметру невозможно. В языке Haskell с этим никаких проблем нет, в этом случае просто порождается функция с меньшим на единицу числом параметров. Отказ от скобок при вызове функций в языке Haskell имеет глубокий смысл. Он позволяет интерпретировать применение функции к нескольким параметрам как применение функции последовательно к каждому параметру по одному: $f\ x\ y \equiv (f\ x)y$.

Таким образом, важной является возможность применения функции к одному (первому) параметру. При этом должна порождаться функция с меньшим на единицу числом параметров, которую, в свою очередь, можно применить к ее первому параметру (второму параметру исходной функции) и т.д. В библиотеке funcprog для этой цели был выбран оператор <<. Он является весьма точным аналогом оператора $ языка Haskell. В частности, если применяемая функция имеет один параметр, то в результате получится функция без параметров. Тип значения, к которому применяется функция, должен соответствовать типу первого параметра функции. Оператор << определен следующим образом:

```
template<typename Ret , typename Arg0 , typename . . . Args>
function_t<Ret (Args . . . )> operator<<(
  function_t<Ret (Arg0 , Args . . . )>const& f ,
  fdecay<Arg0> const& arg0 )
{
  return [ f , arg0 ] ( Args . . . args ) {
    return f ( arg0 , args . . . ) ; } ;
}
```

Это определение относительно несложное. По сути, делается привязка (bind) первого параметра функции. В стандартной библиотеке языка C++ есть функция std::bind, делающая практически то же самое. Проблема с этой функцией состоит в том, что ее результат (функциональный объект) трудно преобразовать в объект std::function из-за того, что функциональный оператор этого объекта шаблонный.

3.5. Композиция функций

В языке Haskell для композиции функций применяется оператор . (точка). В библиотеке funcprog для этого был выбран оператор &:

```
( g & f ) (x , a r g s . . . ) =
  g ( i n v o k e _ f 0 ( f << x ) , a r g s . . . )
```

Здесь x – первый параметр функции f, а *args* – дополнительные параметры функции g. Тип результата функции f должен соответствовать (быть преобразуем к) типу первого параметра функции g. Результатом композиции функций будет функция, имеющая то же число параметров, что и функция g, но тип первого параметра этой функции будет тот же, что и тип первого параметра функции f. Если функция f имеет более одного параметра, то функция g должна первым параметром принимать функцию (результат применения функции f к ее первому параметру).

4. ФУНКТОРЫ, АППЛИКАТИВНЫЕ ФУНКТОРЫ, МОНАДЫ

4.1. Функторы

Функторы и монады в функциональном программировании обычно рассматриваются либо как контейнеры, либо как контексты вычислений. Примеры функторов – списки и класс Maybe. Выход из такого контейнера или контекста, как правило, невозможен, т.е. результатом любых вычислений со значениями внутри функтора или монады является некоторое значение в том же функторе или монаде. Функтор или монада могут содержать любое количество значений, в частности, могут их не содержать вообще (пример – список). Поэтому извлечь явно значение из функтора или монады невозможно. Если стоит задача применить некоторую функцию (с одним аргументом) к значению, хранящемуся в контейнере, то сделать это может только сам контейнер, только он знает, сколько в нем хранится значений и как их извлечь. Для этого предназначен специальный класс *Functor*. В нем есть специальная функция *fmap*, осуществляющая такое применение. Этой же цели служит оператор (⟨$⟩) (в библиотеке funcprog – operator/), синоним функции *fmap*. Каждый контейнер может реализовать свою специализацию класса *Functor* со своей реализацией функции *fmap*. Вот определение класса *Functor*:

```
class Functor f where
  fmap : : ( a −> b) −> f a −> f b
```

Прототип функции *fmap* можно записать в другом эквивалентном виде:

```
fmap : : ( a −> b) −> ( f a −> f b)
```

Таким образом, функцию *fmap* можно рассматривать как функцию с одним параметром, принимающим функцию, принимающую и возвращающую обычные значения и преобразующую ее в функцию, принимающую и возвращающую функторы.

Вот так выглядит специализация класса *Functor* для списков:

```
instance Functor [ ] where
fmap _ [ ] = [ ]
fmap f ( x : xs ) = f x : fmap f xs
```

Результатом применения функции к списку является список из преобразованных значений (возможно, другого типа).

Но если стоит задача применить функцию с двумя аргументами к двум контейнерам (например, просуммировать два списка), то функционала класса *Functor* будет недостаточно. Для решения этой задачи предназначен другой класс – аппликативный функтор (аппликатив). Вот определение класса *Applicative*:

```
class Functor f => Applicative f where
  pure : : a−>f a
  (<*>) : : f ( a−>b)−>f a−>f b
  liftA2 : : ( a−>b−>c )−>f a−>f b−>f c
  liftA2 f x y = f <$> x <*> y
```

Оператор (⟨*⟩) (в библиотеке funcprog operator*) принимает в качестве первого параметра функцию, помещенную в функтор и второго параметра – значение, помещенное в тот же функтор и возвращает результат в том же функторе. Если мы теперь посмотрим на прототип и реализацию функции *liftA2*, то мы увидим, что она передает функцию с двумя параметрами в оператор (⟨$⟩), который принимает функцию с одним параметром. Но противоречия тут нет, так как функцию с прототипом a->b->c мы можем записать так: a->(b->c), т.е как функцию с одним параметром, возвращающую функцию. Тогда оператор (⟨$⟩) нам как раз и вернет функцию (b->c), помещенную в функтор, которая затем передается в оператор (⟨*⟩). По аналогии с функцией *fmap*, прототип функции *liftA2* мы можем записать так:

```
liftA2 : : ( a−>b−>c )−>( f a−>f b−>f c )
```

т.е. рассматривать ее как функцию с одним аргументом, принимающую функцию, работающую с обычными значениями и возвращающую функцию, работающую с функторами, “поднимающую” функцию с двумя аргументами (отсюда и ее название) в аппликатив. По аналогии с функцией *liftA2* можно написать функцию, “поднимающую” в аппликатив функцию с тремя (и любым другим числом) аргументами:

```
liftA3 : : Applicative f =>
   ( a−>b−>c−>d)−>f a−>f b−>f c−>f d
liftA3 f x y z = f <$> x <*> y <*> z
```

В классе *Applicative* есть также функция *pure*, помещающая обычное значение в “чистый” аппликатив. С ее помощью можно, например, написать функцию, “поднимающую” в аппликатив функцию с одним параметром:

```
liftA : : Applicative f=>( a−>b)−>f a−>f b
liftA f x = pure f <*> x
```

4.2. Монады

Монады можно рассматривать как дальнейшее продолжение аппликатива, они предназначены для построения цепочек монадных вычислений. Каждая монада имеет две основных операции: *mreturn* (в языке Haskell *return*) и *mbind* (в языке Haskell и в библиотеке funcprog оператор ⩾=). Операция *mreturn* аналогична операции *pure* для аппликативов (фактически для большинства монад *mreturn* определяется как *pure*), а операция *mbind* (>>=) имеет следующее определение:

```
(>>=) : : (Monad m) −> m a −> ( a −> m b) −> m b
```

Она принимает в качестве параметров монаду и функцию, принимающую обычное (не монадное) значение и возвращающую монадное значение (возможно, другого типа, но в той же монаде). Если исходная монада пустая, то возвращается также пустая монада, иначе значение извлекается из монады, к нему применяется функция и возвращается ее результат.

Для каждой монады две монадные операции должны удовлетворять трем т.н. “монадным законам”. Для того, чтобы их сформулировать, введем операцию монадной композиции функций (*mcompose* или оператор (>=>)). Она определяется следующим образом:

```
(>=>) : : f >=> g = \x −> ( f x >>= g )
```

Оба операнда – функции, принимающие обычные значения и возвращающие монадные. Результатом монадной композиции функций также является функция, принимающая обычное значение и возвращающее монадное. Следовательно, опера-

цию монадной композиции можно рассматривать как групповую операцию в пространстве таких функций (принимающих обычное значение и возвращающих монадное в некоторой монаде). В терминах этой групповой операции монадные законы формулируются так:

```
1. mreturn >=> f == f
2. f >=> mreturn == f
3. (f >=> g) >=> h == f >=> (g >=> h)
```

Другими словами, монадные операции *mreturn* и *mbind* должны быть определены так, чтобы, во-первых, операция *mreturn* являлась единичным элементом (левым и правым) монадной композиции (первые два закона), и, во-вторых, монадная композиция должна быть ассоциативной (третий закон).

Еще одна очень широко применяемая при работе с монадами конструкция – т.н. *do*-нотация. Она позволяет существенно сократить запись и сделать ее более наглядной. Смысл ее в следующем. Пусть *m* – некоторое монадное значение, а *action* – некоторое вычисление, которое нужно провести со значением, хранящимся в *m* (и которое, естественно, возвращает монадное значение в той же монаде). Это можно было бы записать на языке Haskell следующим образом:

```
m >>= \x -> action
```

С помощью *do*-нотации то же самое записывается так:

```
do x <- m; action
```

Если нужно извлечь значения из двух монад, то разница еще существеннее. Вместо

```
m1 >>= \x -> m2 >>= \y -> action
```

можно написать:

```
do x <- m1; y <- m2; action
```

Библиотека funcprog также поддерживает *do*-нотацию. Она реализована с помощью макроса *_do*:

```
#define _do( var , mexpr , expr ) \
  ( (mexpr ) >>= _( [=] ( typename \
    std::decay_t<decltype (mexpr )> \
   ::value_type const& var ) { expr }) )
```

С помощью этого макроса описанное выше вычисление можно записать так:

```
_do(x , m, action )
```

Для большего числа переменных (от двух до пяти) есть макросы от *_do2* до *_do5*, реализованные через цепочку обращений к макросу *_do*. Вот текст реализации макросов *_do2* и *_do3*:

```
#define _do2( v1 ,m1 , v2 ,m2, expr ) \
  _do( v1 ,m1 , return _do( v2 ,m2, expr ) ; )

#define _do3( v1 ,m1, v2 ,m2 , v3 ,m3, expr ) \
 _do2( v1 ,m1, v2 ,m2,
   return _do( v3 ,m3, expr ) ; )
```

Соответственно, пример с двумя монадами можно записать так:

```
_do2(x , m1, y , m2, action )
```

4.3. Реализация функторов и монад

С помощью библиотеки funcprog любой подходящий класс (в том числе и внешний) можно сделать функтором и монадой. В сам класс никаких изменений вносить не требуется. В рамках самой библиотеки монадами являются, к примеру, классы *Identity*, *Maybe* и *List*. Рассмотрим общие принципы реализации. Любой функтор или монада являются типом с одним параметром. В языке C++ типы с параметром реализуются с помощью шаблонов классов. Например, шаблон класса *Maybe* определен так:

```
template<typename A> struct Maybe;
```

Но сам шаблон класса типом не является, и его нельзя передать как параметр шаблона другого класса, что бывает иногда нужно. Чтобы это можно было сделать, определяется еще один класс (без шаблона) с именем *_Maybe* (с подчерком впереди). Это уже настоящий класс, его можно передавать как параметр шаблона. В нем определена метафункция *type*, возвращающая класс *Maybe*:

```
struct _Maybe {
  template<typename A>
  using type = Maybe<A>;
};
```

Шаблон класса *Maybe* наследуется от этого класса:

```
template<typename A>
struct Maybe : _Maybe , optional_ t <A>{
  ...
};
```

Для того, чтобы класс *Maybe* сделать функтором, нужно написать специализацию шаблонов классов *is_functor* и *is_same_functor* для класса *Maybe* и специализацию шаблона класса *Functor* для класса *_Maybe*. Шаблон класса (метафункции) *is_functor* принимает тип и возвращает в переменной *value* булевское значение, говорящее о том, является этот тип функтором. Шаблон класса (метафункции) *is_same_functor* принимает два типа и возвращает булевское значение, говорящее о том, являются эти типы одним и тем же функтором. По умолчанию обе эти метафункции всегда возвращают значение *false*:

```
template<typename T>
struct is_functor : std::false_type {};
template<typename T1 , typename T2>
struct is_same_functor : std::false_type {};
```

Шаблон класса *Functor* просто продекларирован:

```
template<typename T> struct Functor ;
```

Специализация этих классов для класса *Maybe* выглядит так:

```
template<typename A>
struct is_functor<Maybe<A> > :
  std::true_type {};
template<typename A1 , typename A2>
struct is_same_functor<Maybe<A1>,
  Maybe<A2> > : std::true_type {};
template<> struct Functor<_Maybe>{
  ...
};
```

Внутри класса Functor<_Maybe> требуется определить метафункцию *fmap_result_type* и статическую функцию *fmap*. Метафункция *fmap_result_type* принимает в качестве параметра тип функции – первого параметра функции *fmap* и возвращает в переменной *type* тип возвращаемого функцией *fmap* значения.

Определение аппликатива и монады аналогично определению функтора. Нужно создать специализации шаблонов классов *is_applicative*, *is_same_applicative* и *Applicative* для аппликатива и шаблонов классов *is_monad*, *is_same_monad* и *Monad* для монады. Для аппликатива в специализации шаблона класса *Applicative* нужно определить статические функции *pure* и *ap*, а также метафункцию *ap_result_type*, вычисляющую тип результата функции *ap*. Для монады в специализации шаблона класса *Monad* нужно определить статические функции *mreturn* и *mbind*, а также метафункцию *mbind_result_type*, вычисляющую тип результата функции *mbind*.

5. ПОЛУГРУППЫ, МОНОИДЫ

Полугруппа (Semigroup) в общей алгебре – множество с заданной на нем ассоциативной бинарной операцией. В языке Haskell эта операция задается оператором (⟨ ⟩), а в библиотеке funcprog – оператором %. Моноид – это полугруппа с единичным элементом. Единичный элемент моноида возвращается функцией *mempty*. Аналогично функторам и монадам, для того, чтобы некоторый класс сделать полугруппой и моноидом, нужно создать специализации шаблонов классов *is_semigroup*, *is_same_semigroup* и *Semigroup* для полугруппы и шаблонов классов *is_monoid*, *is_same_monoid* и *Monoid* для моноида. Для полугруппы в специализации класса *Semigroup* нужно создать статические функции *semigroup_op* и *stimes*, а для моноида – *mempty* и *mappend*.

Аналогично функторам, монадам, полугруппам и моноидам реализованы классы *Alternative* (моноиды над аппликативами) с операциями *empty* и (⟨|⟩) (в библиотеке оператор |) и *MonadPlus* – монады с поддержкой выбора и неудачи (failure) с операциями *mempty* и *mplus*.

6. FOLDABLE И TRAVERSABLE

6.1. Класс Foldable

Класс *Foldable* является обобщением сверточного функционала, первоначально определенного для списков, на произвольные типы данных, поддерживающих свертку. Основные сверточные операции – это правая и левая свертки (*foldr* и *foldl*). Эти функции принимают в качестве параметров функцию, по которой делается свертка, начальное значение и свертываемый объект (реализующий класс *Foldable*). Если тип данных, хранящихся в свертываемом объекте, является моноидом, то для такого объекта в классе *Foldable* определена также функция *fold*, принимающая единственный параметр – свертываемый объект. Этот объект сворачивается в моноид, например, список списков свернется в один список, включающий все объекты из всех списков. В качестве начального значения при свертке берется единичный элемент моноида (возвращаемый функцией *mempty*), а в качестве сверточной функции – функция *mappend*. Так как моноидная операция ассоциативна, то правая и левая свертки дают одинаковый результат.

Еще одна важная функция класса *Foldable* – *foldMap*. Ее прототип на языке Haskell выглядит так:

```
foldMap : ( Foldable t , Monoid m) =>
    ( a -> m) -> t a -> m
```

Эта функция принимает в качестве параметров функцию (в свою очередь принимающую в качестве параметра значение того типа, который хранится в свертываемом объекте и возвращающую моноид) и свертываемый объект и возвращающую моноид. Как и функция *fold*, она делает свертку объекта в моноид. Значения, хранящиеся в свертываемом объекте (произвольного типа), предварительно пропускаются через функцию, принимаемую в качестве первого параметра (и, таким образом, преобразуются в моноид). Функцию *fold* можно естественным образом определить через функцию *foldMap*:

```
fold = foldMap id
```

Здесь id – функция, возвращающая свой аргумент: ($idx = x$), или, на C++:

```
template<typename T>
T id (T const& value ) { return value ; }
```

В последнем определении функции *fold* использовалась упоминавшаяся выше η-редукция. Полное определение выглядит, естественно, так: $foldx = foldM\ ap\ id\ x$. Но в языке Haskell параметры, стоящие в самом конце левой и правой частей определения функции, можно опускать, если они совпадают. Библиотека funcprog, к сожалению, η-редукцию не поддерживает, поэтому определение функции fold в библиотеке выглядит так:

```
template<typename F>
typename F::value_type fmap (F const& x ) {
 return foldMap (
   _(_id<typename F::value_type>) , x ) ;
}
```

Смотрится более громоздко, но видно, что написано в принципе то же самое, просто типы приходится указывать явно, что удлиняет запись.

Функция *foldMap* похожа на функцию *fmap* (не случайно сходство их названий). Это видно из сравнения определений этих функций для списков:

```
instance Functor [ ] where
   fmap _ [ ] = [ ]
   fmap f ( x : xs ) = f x : fmap f xs

instance Foldable [ ] where
   foldMap _ [ ] = mempty
   foldMap f ( x : xs ) = f x <> foldMap f xs
```

Обе функции проходят по списку, при этом функция *fmap* формирует результат, перестраивая список, а *foldMap* – свертывая значения в моноид с помощью функции *mappend* (оператора $\langle\ \rangle$).

6.2. Класс Traversable

Класс *Traversable* играет для аппликативов ту же роль, что класс *Foldable* для моноидов. Основная функция класса *Traversable* – *traverse*. Вот ее прототип:

```
traverse ::Applicative f=>
( a -> f b) -> t a -> f ( t b)
```

Функция, передаваемая первым параметром, возвращает значение в аппликативе, значит, и функция *traverse* должна вернуть результат в том же аппликативе (выход за пределы функтора запрещен). Определение этой функции для списков выглядит так:

```
instance Traversable [ ] where
  traverse _ [ ] = pure [ ]
  traverse f ( x : xs ) = ( : ) <$> f x <*>
     traverse f xs
```

Сравнение этого определения с определением функции *foldMap* класса *Foldable* показывает большое внешнее сходство. Функция *traverse* проходит по элементам списка (в общем случае – по элементам объекта класса *Traversable*), преобразует каждый элемент в аппликатив с помощью функции – первого параметра и формирует новый аппликатив, внутри которого хранится исходная структура объекта, но с новыми значениями. Другими словами, функция *traverse* создает новый внешний (аппликативный) уровень для исходного контекста.

Если значения, хранящиеся внутри объекта класса *Traversable*, являются аппликативами, возможен простой обход данного объекта. Для этого предназначена другая функция класса *Traversable* – *sequenceA*. Вот ее прототип и определение на языка Haskell:

```
sequenceA ::Applicative f=>
   t ( f a ) -> f ( t a )
sequenceA = traverse id
```

В классе *Traversable* есть еще две функции – *mapM* и *sequence*. Они аналогичны функциям *traverse* и *sequenceA* (фактически их вызывают), но работают с монадами.

7. READER, WRITER И STATE

7.1. Монада Reader

Монада *Reader* предназначена для предоставления общей информации (например, общих параметров системы) одновременно нескольким функциям. Альтернативный подход – передавать эту информацию во все функции в качестве параметров или хранить ее в глобальных переменных, оба подхода часто оказываются неудобными. Монада *Reader* позволяет решить эту проблему, предоставляя общий для всех контекст, в котором любая функция может запросить информацию. Тип хранимого значения передается как параметр класса Reader. Основные функции класса *Reader – ask*, возвращающая значение, *local*, временно меняющая контекст и *reader*, создающая объект *Reader*. Вот небольшой демонстрационный пример использования этой монады на языке Haskell:

```
hello = reader $ \name ->
    ("hello , " ++ name ++ "!")
bye = reader $ \name ->
    ("bye , " ++ name ++ "!")
convo = reader ( const (++) ) <*>
    hello <*> bye
main = print . runReader convo $ "Fred"
```

Функция *main* печатает: "Hello,Fred!Bye,Fred!". Обе функции *hello* и *bye* прочитали переданное значение ("Fred"). Вот как выглядит та же программа на C++:

```
using R = String ;
using _Rdr = _Reader<R>;
const Reader<R, String>
 hello=_Rdr::reader (_( [ ] (R const& name ) {
  return String ("Hello , "+name+"!") ; }) ) ,
 bye=_Rdr::reader (_( [ ] (R const& name ) {
  return String ("Bye, "+name+"!") ; }) ) ,
 convo=_Rdr::reader (_const_<String>(
_(concat2<char>) ) ) * hello * bye ;
BOOST_TEST(convo. run("Fred"). run ( ) ==
  "Hello ,Fred!Bye,Fred!") ;
```

Как обычно, на C++ получилось несколько длиннее, но видно, что делается все то же самое.

7.2. Монада Writer

Монада *Writer* предназначена для вывода информации в некоторое общее для всех функций место (например, для логирования). При этом функция не должна заботиться о том, как и куда выводить логи. Если функция оказалась в контексте монады *Writer*, ей становятся доступными функции класса *Writer*. *Writer* выводит информацию в любой моноид, например, в строку (тип моноида передается как параметр класса). Вот пример, демонстрирующий использование монады *Writer* на примере функции, делящей число пополам:

```
half x = do
  tell ("I_just_halved_"++(show x )++"!")
  return ( x `div ` 2)
main = print . execWriter $
  half8 >>= half
```

Функция *main* печатает: "I just halved 8!I just halved 4!". Вот тот же самый пример на C++:

```
using W = String ;
using _Wrt = _Writer<W>;
using mWrt = Monad<_Wrt>;
const function_t<Writer<W, int>( int )>
 half = [ ] ( int x ) { return _Wrt::tell (
 W("I_just_halved_" + eval ( x ) + "!") )
 >> mWrt::mreturn ( x / 2) ; } ;
BOOST_TEST(half ( 8 ) . exec ( ) . run ( )
 == "I_just_halved_8!") ;
BOOST_TEST((half ( 8 )>>=half). exec ( ). run ( )
 == "I_just_halved_8! I_just_halved_4!") ;
```

7.3. Монада State

Монада *State* похожа на монаду *Reader* в том, что она также хранит некоторое значение (состояние), его тип передается как параметр класса. Отличие же состоит в том, что это хранимое значение можно менять. Эту монаду можно, например, использовать в качестве аналога обычной (изменяемой) переменной в императивных языках программирования. Смысл же этого класса в том, чтобы хранить некоторое общее для всех функций состояние, которое все функции могут запрашивать и при необходимости менять. Основные функции класса *State –get*, *put* и *modify*, которые, соответственно, запрашивают состояние, устанавливают новое и модифицируют.

8. ТРАНСФОРМЕРЫ МОНАД

В реальных вычислениях обычно бывает нужно использовать не какую-то одну монаду, а одновременно несколько. Например, все три описанные в предыдущей главе монады – явные кандидаты на одновременное использование. Есть еще монады

Maybe и *Either*, обрабатывающие ошибки вычислений, и другие монады. Поэтому встает вопрос об объединении монад. Этой цели служат т.н. трансформеры монад, позволяющие вносить одну монаду внутрь другой. С помощью трансформеров монад строится стек монад, позволяющий вести вычисления одновременно в нескольких монадах. В настоящий момент в библиотеке funcprog реализованы трансформеры монад *IdentityT*, *MaybeT*, *ReaderT*, *WriterT* и *StateT*. А сами классы *Reader*, *Writer* и *State* реализованы, как и в языке Haskell, как "неподвижные точки" (fixed points) соответствующих трансформеров монад, а именно, каждый из этих классов является соответствующим трансформером монады *Identity*, не меняющей значение. В следующем примере параметр делится на число из монады *Reader*, лог пишется в монаду *Writer*:

```
type Divide=WriterT String (Reader Int)
runApp k n=runReader (runWriterT k) n

divide :: Int -> Divide Int
divide x = do
 n <- ask
 tell $ "Dividing " ++ (show x) ++
   " by " ++ (show n) ++ "!"
 return $ x `div` n

main = print . runApp (divide 8) $ 2
```

Функция *main* печатает: (4,"Dividing 8 by 2!"). Здесь 4 – это результат деления. Монады *Reader* и *Writer* можно поменять местами. При этом изменятся только две первых строки:

```
type Divide=ReaderT Int (Writer String)
runApp k n=runWriter (runReaderT k n)
```

То же самое на языке C++ выглядит так:

```
template<typename _Divide>
typename _Divide::template type<int>
divide (int x){return _do(
   n, MonadReader<int, _Divide>::ask(),
   return MonadWriter<String, _Divide>
    ::tell("Dividing " + eval(x) +
    " by " + eval(n) + "!")
   >> Monad<_Divide>::mreturn(x/n);
 );
}
BOOST_AUTO_TEST_CASE(test_MonadTrans){
  BOOST_TEST(eval(
     divide<_WriterT<String, _Reader<int>>>
      (8).run().run(2).run()) ==
       "(4,\"Dividing 8 by 2!\")");
}
```

При изменении порядка монад *Reader* и *Writer* меняются местами две строки:

```
   divide<_ReaderT<int, _Writer<String>>>
   (8).run(2).run().run()) ==
```

9. КОМБИНАТОРНЫЙ ПАРСЕР PARSEC

В языке Haskell *Parsec* [11] – это простой, хорошо документированный и быстрый парсер. Он определяется как неподвижная точка трансформера монад *ParsecT*. Библиотека *Parsec* представляет из себя набор достаточно простых парсеров с широкими возможностями их комбинирования. Сам класс *ParsecT* является функтором, аппликативом, монадой и моноидом. Соответственно, все возможности, которые предлагают эти классы, для парсеров также доступны. Парсер *Parsec* достаточно полно реализован в библиотеке funcprog.

Главная проблема при его реализации была следующая. Функция *unParser* в определении класса *ParsecT* определена как полиморфная:

```
unParser :: forall b.
      State su
  ... -- Текст опущен
-> m b
```

В терминах языка C++ это означает, что *unParser* – это шаблон функции с параметром шаблона *b*. А такую функцию (*function_t*) описать невозможно. Метод в классе с таким определением нельзя сделать виртуальным. Единственный выход – сделать в классе метод с таким определением и передать тип класса, реализующий этот метод как дополнительный параметр шаблона класса *ParsecT*. Это создало дополнительные трудности при реализации парсера, но в конце концов все получилось. И калькулятор заработал. Опишем вкратце эти трудности.

Класс *ParsecT* в языке Haskell имеет 4 параметра: ParsecT⟨S, U, M, A⟩, где *S* – тип входного потока, *U* – пользовательские данные, *M* – монада и *A* – тип выходных данных. Таким образом, два разных (имеющих разную реализацию функции *unParser*) парсера, у которых эти 4 параметра совпадают, имеют один тип, следовательно, некоторая функция, которая должна вернуть парсер, может, в зависимости от некоторого условия, вернуть один из двух этих парсеров (if cond then p1 else p2). В библиотеке funcprog по указанной выше причине к классу *ParsecT* был добавлен еще один параметр *P*, отвечающий как раз за реализацию метода *unParser*. Таким образом, в библиотеке funcprog разные парсеры всегда имеют разные типы и написать подобную конструкцию оказывается невозможным. Если в исходном тексте на языке Haskell встречается такая конструкция, то такую функцию перенести на C++ не удается. К счастью, такие случаи встречаются нечасто.

10. ЗАКЛЮЧЕНИЕ

Функциональное программирование – мощный инструмент при реализации логически сложных алгоритмов. Изящность и красота языков функционального программирования, таких, как Haskell, основаны на возможности комбинирования функций, когда сложное строится из простого. Язык C++, хотя и содержит в себе элементы функционального программирования (например, лямбда-выражения), в полной мере такими возможностями не обладает. Но опыт написания библиотеки funcprog показал, что язык C++ достаточно мощный для того, чтобы средствами библиотеки шаблонов функций и классов реализовать многие из возможностей языка Haskell. В частности, удалось достаточно полно реализовать парсер *Parsec*.

Функциональный стиль программирования позволяет упростить запись логически сложных алгоритмов за счет того, что такой стиль позволяет “поднять” запись алгоритмов на более высокий логический уровень, что делает эту запись более понятной и одновременно менее громоздкой. Функциональное программирование широко используется во многих областях программирования, в частности, для обработки текстов (например, при компиляции). Данная библиотека ставит цель привнести этот стиль программирования в численные методы, в частности, в планах – встроить библиотеку funcprog в разработанный автором сеточно-операторный подход к программированию (см. [12, 13]), что позволит создавать сложные операторы путем комбинирования более простых, легко “превращать” обычные функции в сеточные (свойство функторов) и т.д. Исследование влияния применения этого стиля на быстродействие (что является чрезвычайно важным в численных методах) пока полноценно не проводолись. Ясно, что замедление будет (привнесение нового уровня абстракции к этому приводит неизбежно), но важным является не сам этот факт, а численные оценки. Замедление в разы, конечно же, неприемлемо, но автор надеется, что это будут скорее проценты, и плата за несомненное удобство не будет слишком большой.

СПИСОК ЛИТЕРАТУРЫ

1. Haskell language. https://www.haskell.org/
2. The Scala Programming Language. https://www.scala-lang.org/
3. Standard ML of New Jersey. http://smlnj.org/
4. Visual F#. https://msdn.microsoft.com/ru-ru/visualfsharpdocs/conceptual/visual-fsharp
5. FC++: Functional Programming in C++. https://yanniss.github.io/fc++
6. C++ template library for fans of functional programming. https://github.com/beark/ftl
7. *David Abrahams*, *Aleksey Gurtovoy*. C++ Template Metaprogramming. Addison-Wesley. 2004.

8. *Краснов М.М.* Метапрограммирование шаблонов C++ в задачах математической физики. М.: ИПМ им. М.В. Келдыша, 2017. 84 с. DOI: http://keldysh.ru/e-biblio/krasnov https://doi.org/10.20948/mono-2017-krasnov

9. *Краснов М.М., Ладонкина М.Е.* Разрывный метод Галеркина на трехмерных тетраэдральных сетках. Применение шаблонного метапрограммирования языка C++. Программирование, 2017 г., № 3, стр. 41–53.https://elibrary.ru/download/elibrary_29207399_42944924.pdf

10. Boost C++ Libraries. https://www.boost.org/

11. parsec: Monadic parser combinators. http://hackage.haskell.org/package/parsec

12. *Краснов М.М.* Операторная библиотека для решения трехмерных сеточных задач математической физики с использованием графических плат с архитектурой CUDA. Математическое моделирование, 2015, т. 27, № 3, стр. 109–120. http://www.mathnet.ru/links/38633e7a627ab2ce1527ae4a092be72f/mm3585.pdf

13. *Краснов М.М.* Кандидатская диссертация “Сеточно-операторный подход к программированию задач математической физики”. Автореферат. http://keldysh.ru/council/1/2017-krasnov/avtoref.pdf

ПРОГРАММИРОВАНИЕ, 2020, № 5, с. 60–71

БАЗЫ ДАННЫХ, ХРАНИЛИЩА ДАННЫХ

УДК 681.3

РЕПЛИКАЦИЯ В РАСПРЕДЕЛЕННЫХ СИСТЕМАХ: МОДЕЛИ, МЕТОДЫ И ПРОТОКОЛЫ

© 2020 г. А. Р. Насибуллин[a],*, Б. А. Новиков[b],**

[a] *Санкт-Петербургский государственный университет*
199034 Санкт-Петербург, Университетская набережная, д. 7/9, Россия

[b] *Национальный исследовательский университет "Высшая школа экономики"*
190121 Санкт-Петербург, ул. Союза Печатников, д. 16, Россия

**E-mail: nevskyarseny@yandex.ru*

***E-mail: bnovikov@hseu.ru*

Поступила в редакцию 10.03.2019 г.
После доработки 20.09.2019 г.
Принята к публикации 25.10.2019 г.

Репликация данных применяется для повышения надежности, доступности и пропускной способности систем хранения данных за счет увеличения сложности и стоимости обновления данных. Во многих случаях в системах хранения данных с репликацией применяются ослабленные критерии согласованности. В обзоре рассматриваются различные схемы репликации данных; обсуждаются разнообразные модели и протоколы, обеспечивающие согласованность.

DOI: 10.31857/S0132347420050064

1. ВВЕДЕНИЕ

Статья посвящена обсуждению различных моделей поддержки согласованности и протоколов репликации данных. *Репликация данных* – это способ организации хранения данных, в котором каждый элемент данных хранится в нескольких копиях, размещенных на разных серверах. Каждая из таких копий называется *репликой*. Задача репликации – защита от потери данных. Репликация данных обладает следующими свойствами:

• *Доступность*. Обеспечивается возможность доступа к данным в случае, если часть реплик недоступна.

• *Надежность*. Предотвращается потеря данных при частичном разрушении серверов или потере части реплик.

• *Пропускная способность*. Увеличивается пропускная способность чтения данных.

В статье будут использованы следующие количественные характеристики баз данных: пропускная способность, масштабируемость. *Пропускная способность* системы определяется как количество выполняемых действий разной сложности за единицу времени. *Масштабируемость* определяется следующей формулой:

$$S = \frac{M}{N}, \qquad (1)$$

где M означает пропускную способность базы данных, состоящей из множества вычислителей, с определенным объемом данных и определенной нагрузки. Под N подразумевается пропускная способность системы, состоящей из одного вычислителя.

Введем определение распределенной базы данных. Под *распределенной базой данных* понимается система хранения данных, состоящая из локальных баз данных, размещенных в различных узлах компьютерной сети. Каждая из локальных баз данных может иметь реплику.

При использовании репликации может возникать проблема поддержки всех реплик в согласованном состоянии [1–5]. Другими словами, результат выполнения запроса может отличаться в зависимости от того, с какой репликой работает клиент. Клиентский запрос, выполняемый на одной из реплик, может обрабатывать неакутальные данные. При этом, поддержка согласованности на всех репликах может привести к ухудшению производительности распределенной базы данных и сокращению количества обрабатываемых клиентских запросов, а также возникновению проблем масштабирования.

Традиционно согласованность рассматривается в контексте СУБД как инструмент для соотнесения значений разных элементов данных и реализуется средствами управления транзакциями. Неформально транзакции принято характеризо-

вать свойствами ACID, а степень согласованности описывается в терминах уровней изоляции. Такое понимание согласованности абстрагировано от архитектуры системы (централизованной или распределенной) и основано на предположении о *единой логической копии* каждого элемента данных.

В распределенных системах с репликацией такое понимание согласованности оказывается недостаточным и дополняется рассмотрением согласованности нескольких копий одного элемента данных и согласованности реплик. В литературе по распределенным системам (см., например, [6]) обычно понятие транзакций явно не упоминается, все операции обновления считаются атомарными и предполагается, что локальная согласованность каждой реплики сохраняется при обновлениях, а для достижения глобальной согласованности данных достаточно обеспечить согласованность (то есть идентичность) копий.

Однако буквальная реализация концепции единой логической копии зачастую оказывается слишком дорогостоящей, поэтому в большинстве моделей, обсуждаемых далее в разделе 3, применяются ослабленные критерии согласованности. В нашей работе мы рассматриваем модели и протоколы согласованности копий необязательно использующие механизмы транзакций. Далее в этой работе рассматривается согласованность реплик, если явно не оговорено другое.

В последнее время исследователи предложили различные решения задач масштабирования и поддержки согласованности данных. Например, в работах [7, 8] предлагается ослабленная модель согласованности, обеспечивающая высокую пропускную способность системы и упрощающая задачу масштабирования. В работах [5, 9] предлагается более усиленная модель согласованности, чем упомянутая выше.

Оставшаяся часть статьи построена следующим образом. В разделе 2 проведен анализ различных схем репликации. В разделе 3 рассматриваются модели согласованности. Одной из важных задач, обсуждаемых в разделе 4, является задача по распространению данных от одной реплики к другим. Также, в разделе 4 представлено краткое сравнение различных подходов доставки данных к серверам. В разделе 5 затронуты различные протоколы согласованности. Краткий обзор алгоритмов репликации изложен в разделе 6. В разделе 7 подведены итоги данной статьи.

2. СХЕМЫ РЕПЛИКАЦИИ

В этом разделе мы обсудим два типа схем репликации – статические и динамические [10, 11].

Схема называется статической, если задано фиксированное количество реплик в распределенной базе данных. Число реплик в такой схеме может быть изменено путем изменения конфигурации и рестарта системы. Недостатком данного типа схемы репликации является большая трудоемкость по расширению распределенной базы данных под растущие нужды бизнеса.

В отличие от статических схем, в динамической схеме новые реплики добавляются или удаляются динамически без рестарта системы в зависимости от количества клиентских запросов. Благодаря увеличению количества работающих серверов удается повысить пропускную способность. Администратор баз данных может изменять количество реплик в кластере, позаботившись о синхронизации добавленной реплики с остальными. Динамические схемы особенно полезны в географически распределенных системах, где в каждой локальной зоне может быть создано сколько угодно новых реплик. Но несмотря на это, динамические схемы имеют недостаток – необходимость выполнять дополнительную работу по определению физических адресов других серверов, хранящих реплики, что увеличивает количество сообщений, пересылаемых между серверами и вспомогательными сервисами. Статические схемы лишены данного недостатка [12, 13].

Для статической и динамической схем репликации известны два подхода по распространению обновлений: централизованный и децентрализованный [10, 22]. Достоинством децентрализованного подхода является отсутствие фиксированного сервера, принимающего все запросы от клиентов. Недостатком децентрализованного подхода является необходимость решать проблему синхронизации данных между репликами.

В централизованном подходе выделяется роль основного сервера-координатора, обрабатывающего запросы клиентов. Отказ работы основного сервера приведет к неработоспособности всего кластера. Данная проблема может быть решена с помощью кворумных протоколов [23, 24]: если координатор становится недоступным, оставшиеся доступные сервера выбирают координатора путем голосования. В таблице 1 представлены примеры централизованных и децентрализованных решений на основе статических и динамических схем.

Как в статической, так и в динамической схемах рассматривают два класса стратегий репликации данных, влияющие на организацию копирования данных по всем репликам: полная и частичная [22].

Под полной репликацией подразумевается копирование всех данных, хранящихся в основном сервере, на все сервера. При этом, полная репликация позволяет достичь надежности и высокой

Таблица 1. Примеры централизованных и децентрализованных решений

	Статическая	Динамическая
Централизованная	CitusDB [14], PostgreSQL [15] MySQL [16]	Google Spanner [5] MongoDB [17], Cockroach DB [18]
Децентрализованная	Bigtable [19]	Cassandra [20], Dynamo [21]

доступности распределенной базы данных. Применение подобного подхода приводит к большим затратам, так как каждый сервер должен обладать достаточно большим объемом свободного места на носителях для хранения данных, пересылаемых между серверами по сети.

При частичной репликации, в отличие от полной репликации, каждый сервер кластера базы данных хранит лишь некоторое подмножество данных. Другими словами, для каждого элемента данных основного сервера хранится реплика на нескольких серверах, но не на всех. При отказе большого количества серверов часть данных может стать недоступной. Тем не менее, стоимость хранения данных и коммуникация между серверами кластера значительно меньше, нежели в стратегии полной репликации данных.

3. МОДЕЛИ СОГЛАСОВАННОСТИ

В данном разделе будут рассмотрены модели согласованности, используемые в схемах репликации.

Модель согласованности определяется как контракт между базой данных и клиентами. Если для потока клиентских запросов определен порядок выполнения, то произойдет следующее: данные будут согласованны, клиент получит корректные данные. Различают два семейства моделей согласованности [1]: *клиент-ориентированные модели согласованности* и *модели согласованности ориентированные на данные*.

Для удобства определения моделей согласованности введем следующие обозначения. Предположим, что имеются два клиентских процесса P_1, P_2, и хранилище данных *DS*. Здесь и далее в статье клиентские процессы для базы данных представляют собой транзакции. Будем предполагать, что P_i процесс выполняется на сервере *i*. Под $W(x)a$ мы будем подразумевать операцию записи процессов в область памяти x значения a. $R(x)a$ будет означать операцию чтения процессом значения области памяти x, которое возвращает значение a. Будем подразумевать, что результат выполнения операции записи процессом распространяется на другие сервера в рамках другого процесса. Здесь и далее процесс распространения данных от одного сервера к другим обозначен кривой линией на рисунках.

3.1. Модели согласованности ориентированные на данные

В *моделях согласованности ориентированных на данные* акцент делается на том, что результат выполнения запроса на обновление одного сервера немедленно перенаправляется на другие сервера. В этом классе моделей согласованности предполагается несколько одновременно работающих процессов обновления данных.

Известны несколько разновидностей моделей согласованности ориентированных на данные. Далее они перечислены в убывающем порядке по степени строгости согласованности данных.

3.1.1. Внешняя согласованность. Модель внешней согласованности является самой строгой моделью согласованности. Более детально модель внешней согласованности рассматривается в работе [25]. В данной модели выполняется следующее правило: *любая операция чтения области памяти x получает значение последней успешной операции записи в x*. На рисунке 1 процесс P_2 читает значение в области памяти x успешно записанное процессом P_1. Рассматриваемая модель согласованности используется в распределенной базе данных Google Spanner [5].

3.1.2. Последовательная согласованность. Последовательная модель согласованности основана на следующей идее: операции процессов, запущенных на разных серверах, могут чередоваться. При этом операции каждого отдельного процесса

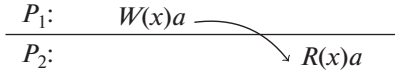

Рис. 1. Внешняя согласованность.

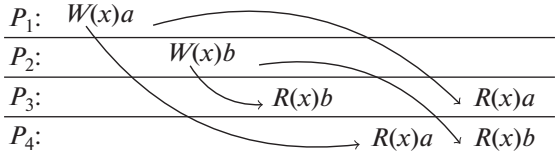

Рис. 2. Последовательная согласованность.

должны быть видны в этой последовательности в порядке, в каком они были выполнены в рамках процесса. Все процессы должны видеть измененные значения данных в одном и том же порядке. В отличие от внешней согласованности, в последовательной модели согласованности не требуется синхронизация глобального времени между всеми процессами, что ослабляет согласованность данных. Как показано на рисунке 2, для P_3 сначала были получены данные от выполнения операции процесса P_2, а потом P_1. Для P_4 были выполнены операции процесса P_1, а затем P_2, что нарушает последовательную согласованность.

3.1.3. Ситуационно-зависимая согласованность. Согласованность данной модели определяется соблюдением следующего условия частичного порядка выполнения операции процессов: выполнение операций процессов P_1, P_2 видны всем процессам в одинаковом порядке в том случае, если операция записи области памяти x процессом P_1 произошла до выполнения операции чтения/записи области памяти x процессом P_2, и выполнение операции чтения/записи процессом P_2 зависит от выполнения операции записи процессом P_1. Такой частичный порядок выполнения операций разных процессов может быть достигнут, например, с помощью часов Лампорта [6]. Если операции чтения/записи процессов P_1, P_2 работают с разными областями памяти, то такие процессы не являются взаимосвязанными и могут быть выполнены на разных серверах в разном порядке, что не допустимо во внешней модели согласованности. Один из примеров реализации ситуационно-зависимой согласованности представлен в работе [26]. На рисунке 3 изображен пример, иллюстрирующий рассматриваемую модель согласованности: операция $W(x)b$ процесса P_2 зависит от выполнения операции $W(x)a$ процесса P_1. Процессы P_3 и P_4 видят одинаковый порядок операции процессов. Операции $W(x)c$ и $W(x)b$ процессов P_1, P_2 являются параллельными.

3.1.4. PRAM(FIFO)-согласованность. PRAM (FIFO)-согласованность определяет следующее правило: операции записи, выполненные процессом на одном сервере, должны быть видны всем другим процессам в том порядке, в котором они были выполнены. При этом операции записи разных процессов могут быть видны в различном порядке на разных серверах.

3.2. Клиент-ориентированные модели согласованности

Если в *моделях согласованности ориентированных на данные* делается акцент на согласованность данных в хранилище между всеми репликами, то в клиент-ориентированных моделях не требуется, чтобы все реплики находились в актуальном состоянии.

Класс клиент-ориентированных моделей согласованности допускает работу клиентских процессов с неактуальными данными на серверах в силу того, что актуальные данные не были распространены по всем серверам. Такое послабление согласованности данных позволяет повысить доступность серверов и увеличить пропускную способность. В статье [27] формально определены модели данного класса.

Для определения разновидностей клиент-ориентированных моделей согласованности потребуются следующие обозначения. Предполагаем, что клиентская согласованность рассматривается в рамках одного процесса P_i. Под L_1, L_2 будем подразумевать

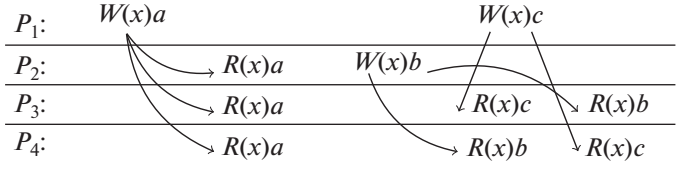

Рис. 3. Ситуационно-зависимая согласованность.

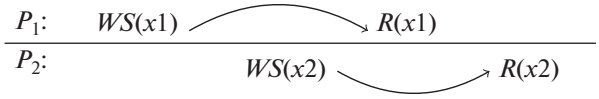

Рис. 4. Монотонное чтение.

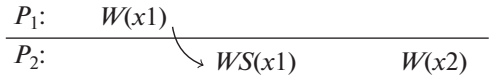

Рис. 5. Монотонная запись.

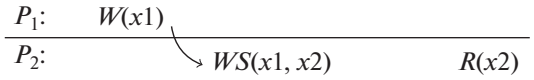

Рис. 6. Читай то, что записал.

реплики, размещенные на разных физических серверах, одного и того же хранилища. Процесс P_i выполняется на реплике L_i. Запись $R(x_j)$ означает операцию чтения данных в области памяти x_j. Операция $WS(x_1, x_2, .., x_n)$ означает множество локальных операций записи некоторого сервера и операций записи, полученных от других серверов в ходе распространения результатов операций записи.

3.2.1. Согласованность в конечном счете. Как утверждается в [28], реплицированные данные, обновленные на одном сервере, в конечном счете обновятся на остальных серверах. Если нет новых обновлений со стороны клиентов, то реплики перейдут в согласованное состояние. При частом обновлении состояние данных в серверах может различаться, что может привести к чтению несогласованных данных из разных серверов [7]. Модель согласованности в конечном счете допускает возникновение аномалии потерянного обновления. Данная модель реализована в отказоустойчивых системах с ленивой моделью репликации [20, 21, 29].

3.2.2. Монотонные чтения. Хранилище реализует модель согласованности монотонного чтения [30] при выполнении следующего условия: если процесс P_i прочитал значение $R(x)$, то, при последующих обращениях к этим данным, процесс получит это же или более новое значение.

На рисунке 4 процесс P_1 пишет в одну реплику значение x_1 и через некоторое время читает его. Спустя некоторое время на вторую реплику распространяется значение x_1 и после чего процесс P_2 пишет во вторую реплику другое значение x_2, а затем в результате чтения на этой реплике клиент гарантированно получит более свежее значение x_2.

3.2.3. Монотонные записи. Идея данной модели [30, 31] заключается в следующем: все операции записи упорядочены в рамках процесса и в таком же порядке распространяются на другие сервера. Как показано на рисунке 5, клиент выполняет на второй реплике операцию записи значения в область памяти x_2 в том случае, если значение в области памяти x_1 успешно распространилось с первой реплики на вторую. В случае, если значение в x_1 не распространилось на вторую реплику и при этом клиент пишет во вторую реплику значение в x_2, согласованность данной модели нарушается.

В отличие от модели *монотонного чтения*, в которой делается акцент на операции чтения в рамках одного процесса, в модели согласованности *монотонной записи* делается акцент на операции записи того же самого процесса.

3.2.4. Читай то, что записал. Как утверждается в [30, 31], это модель, в которой процесс P_i читает на реплике результат последней успешно завершенной операции записи, выполненной на другой реплике тем же процессом. Данная модель согласованности является частным случаем ситуационно-зависимой согласованности. Согласно [27], модель согласованности *читай то, что записал* является более сильной моделью по сравнению с моделью согласованности *монотонного чтения*. Это обусловлено тем, что в модели *читай то, что записал* результат операции записи процессом немедленно распространяется на сервера, на которых этот же процесс выполняет операцию чтения. Данное условие опущено для модели согласованности *монотонного чтения*. Как показано на рисунке 6, процесс P_1 пишет значение, а затем процесс P_2 сначала выполняет операцию $WS(x1, x2)$ и следом опе-

Таблица 2. Пример различных решений, реализующих асинхронную и синхронную модели

	Основной сервер	Любой сервер
Синхронная модель	CitusDB [14], PostgreSQL [15]	Dynamo [21]
Асинхронная модель	MongoDB [17]	Cassandra [20]

рацию чтения, возвращающую актуальное значение x_2.

3.2.5. Запись следует за чтением. Согласно [27, 30, 31], это модель согласованности, в которой процесс P_i производит запись нового значения в область памяти x некоторого сервера только после операции чтения значения x. Данная модель является более сильной моделью согласованности по сравнению с моделью *монотонная запись*, согласно [27]. Это обусловлено тем, что в модели *запись следует за чтением* для новой операции записи в рамках процесса P_i на другом сервере важен результат записи операции в область памяти x, выполненный процессом на другом сервере.

Достоинством таких моделей согласованности, как *монотонная запись*, *монотонное чтение* и *читай то, что записал*, является относительная простота реализации [30, 31]. Если клиент работает с одним сервером, данные модели согласованности гарантируют согласованность данных. В случае, если изначальная сессия на одном сервере оборвалась и клиент подключился на другой сервер, то возникает проблема синхронизации данных между серверами и, как результат, чтение несогласованных данных.

Все перечисленные модели согласованности клиент-ориентированного класса в конечном счете распространяют значения от одного сервера к другим. Когда говорят о согласованности в конечном счете, обычно подразумевают использование связи двух моделей данного класса – *монотонная запись* и *читай то, что записал* [27].

4. ПРОТОКОЛЫ СИНХРОНИЗАЦИИ СЕРВЕРОВ

Важной деталью при реализации алгоритма репликации базы данных является выбор способа копирования данных на все сервера. Согласно [32, 33], существует две модели распространения данных по всем серверам: синхронная модель и асинхронная.

4.1. Синхронная модель

В рамках одной транзакции результат работы, выполненный на одном сервере, фиксируется на других серверах. При этом основной сервер ожидает завершения всех транзакций на остальных серверах. В синхронной модели на всех серверах определена одна и та же модель согласованности, как и на основном сервере. Недостатком синхронной модели является откат глобальной транзакции в случае, если откатывается одна из локальных транзакций. Другим недостатком синхронной модели является высокое время отклика серверов в следствие синхронизации всех реплик в рамках глобальной транзакции при использовании алгоритма двухфазной фиксации [34].

Достоинством данной модели является гарантия согласованности данных на всех серверах. В случае, если основной сервер становится недоступным, клиенты могут быть уверены, что их данные не потеряны, а реплику возможно восстановить из других серверов. Например, синхронная модель реализована в расширении CitusDB [14] для СУБД PostgreSQL. В расширении CitusDB назначается специальный сервер с ролью координатора, который принимает все клиентские запросы и координирует синхронизацию реплик посредством алгоритма двухфазной фиксации [34].

4.2. Асинхронная модель

В асинхронной модели данные фиксируются на основном сервере. После чего выполняются независимые транзакции для обновления данных на других репликах. В отличие от синхронной модели, основной сервер не ожидает выполнения транзакций на остальных серверах и это позволяет увеличить пропускную способность, так как отсутствуют блокирующие операции, в отличие от синхронной репликации. В конечном счете через некоторое время данные на других серверах будут обновлены. Недостатком данной модели является несогласованность данных на разных репликах в некоторый момент времени. Асинхронная модель обеспечивает более высокую доступность системы, чем синхронная. В системах, где производительность системы является важным фактором,

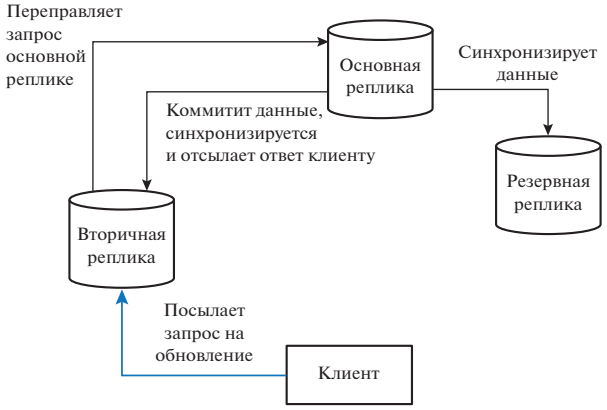

Рис. 7. Протокол удаленной записи.

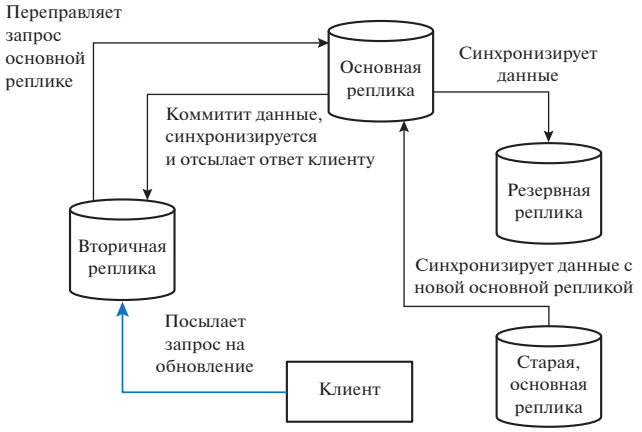

Рис. 8. Протокол локальной записи.

данная модель синхронизации данных остается предпочтительной. Асинхронная модель реализована в таких системах, как [17, 20, 21].

4.3. Сопоставление протоколов синхронизации серверов

В работах [33, 35] рассматриваются распределенные версии асинхронной и синхронной моделей репликации. В таблице 2 представлены различные централизованные и децентрализованные решения, использующие асинхронную и синхронные модели синхронизации данных. В случае с синхронной моделью, обновления могут распространяться от основного сервера к вторичным, где основной сервер играет роль мастер-сервера, обрабатывая все запросы клиентов, блокируя вторичные сервера для синхронизации данных [33]. В качестве альтернативного варианта возможна стратегия, когда клиент посылает запрос всем серверам для обновления данных. В таком случае все транзакции выполняются атомарно [33].

5. ПРОТОКОЛЫ СОГЛАСОВАННОСТИ

В главе 3 мы обсудили различные модели согласованности. В данном разделе будут рассмотренны реализации клиент-ориентированных моделей согласованности и моделей согласованности ориентированных на данные.

5.1. Протоколы на основе первичной копии

Семейство протоколов на основе первичной копии с использованием последовательной модели согласованности работает с упорядоченной последовательностью клиентских запросов. Класс протоколов на основе первичной копии разделен на следующие типы: *протокол удаленной записи*, *протокол локальной записи*, *протокол трансляции журнала транзакций* и *протокол логической репликации*.

В протоколах на основе первичной копии используются следующие принципы работы. Все клиентские запросы обрабатываются специально выделенным сервером называемым *основным* или *ведущим*. Реплика на ведущем сервере называется основной. Сервер, который отслеживает изменения на ведущем сервере, называется *ведомым* или *резервным*. Резервный сервер, к которому нельзя подключаться до тех пор, пока он не будет повышен до ведущего сервера, называется сервером *теплого резерва*. Сервер, который может принимать соединения и обрабатывать клиентские запросы только на чтение, называется сервером *горячего резерва*.

5.1.1. Протокол удаленной записи. Алгоритм протокола удаленной записи [36], изображенный на рисунке 7, состоит из следующих шагов:

1. Основной сервер обрабатывает клиентский запрос и обновляет локально данные.

2. Затем создается запрос на обновление реплики на резервном сервере.

3. Дождавшись ответа от резервного сервера, основной сервер отправляет ответ клиенту.

Если запрос отправляется на один из резервных серверов, то он перенаправляется на основной сервер и запрос проходит все вышеперечисленные шаги протокола.

Недостатком данного подхода является блокирующая операция обновления основного и резервного серверов. Это порождает проблемы с производительностью и доступностью системы. В качестве решения могут быть введены неблокирующие операции [36]. На 1 фазе алгоритма основной сервер отправляет запрос резервному серверу и не дожидается от него ответа. Тем самым увеличивается пропускная способность базы данных.

5.1.2. Протокол локальной записи. В отличие от протокола удаленной записи, в протоколе локальной записи за основу взят подход локального выполнения операции чтения и записи. Такой способ работы характерен для устройств, которые могут работать в оффлайн режиме, например, не подключенные к сети Интернет ноутбуки. В данном алгоритме, устройство может работать со своей копией данных как с основной репликой в оффлайн режиме. Как только устройство становится доступным в сети, происходит процесс синхронизации локальной реплики с основной, как показано на рисунке 8.

5.1.3. Протокол трансляции журнала транзакций. Протокол, реализованный в таких базах данных как PostgreSQL [15] и MySQL [16], довольно прост: ведущий сервер принимает клиентские запросы и заносит запись о результате работы транзакций в журнал транзакций, в то время как каждый резервный сервер работает в режиме приема записей журнала транзакций в виде файлов от ведущего сервера. Если основной сервер отказывает, резервный сервер, содержащий почти все данные с основного сервера, может быть быстро преобразован в новый ведущий сервер. Помимо трансляции журнала транзакций в виде файлов, существует *потоковая репликация*, которая отличается от протокола трансляции журнала транзакций в виде файлов тем, что происходит непрерывная передача записей журнала транзакций резервным серверам в потоковом режиме. Такой подход позволяет работать резервным серверам с меньшей задержкой нежели при передаче файлов журнала транзакций [37].

5.1.4. Протокол логической репликации. Протокол логической репликации появился сравнительно недавно. Реализации протокола встречаются в таких популярных базах данных, как PostgreSQL [15], MySQL [16], OracleDB [38]. В протоколе используется концепция модели читатель-писатель. Основной сервер является писателем, а ведомые – читателями. На стороне писателя определяются данные, которые должны быть распространены на ведомые сервера. Такие данные называются публикациями. Изменения, производимые над публикациями, последовательно распространяются на читателей, тем самым гарантируется согласованность данных.

Недостатком протокола логической репликации является возникновение нарушения согласованности данных в случае, если на стороне читателя выполняются клиентские операции записи. Преимуществом протокола является его относительная простота реализации и возможность выбирать, какие данные могут быть реплицированы.

5.2. Кворумные протоколы

Основная идея кворумных протоколов заключается в следующем: клиент должен получить подтверждение большинства серверов на операции чтения или записи [1–3, 39, 40]. Под кворумом подразумевается количество $\frac{N}{2}+1$ серверов, где N – общее количество серверов. Протоколы на основе кворума широко используются во многих реализациях [5, 41, 42].

Существует две группы кворумных протоколов. Первая группа основана на семействе алгоритмов Paxos. Вторая группа протоколов основана на базе алгоритма Raft. Каждая из групп имеет свои преимущества и недостатки, но их объединяет свойство доступности большинства серверов. Недостатком обеих групп протоколов данного семейства является следующее: в случае, если клиент запрашивает на чтение данные у одного из серверов, то клиент может получить неактуальные данные. Это возможно в силу того, что сервера не успели получить данные от серверов, обладающих свежими данными. Преимуществом является доступность серверов не участвующих в обработке клиентского запроса.

Группа кворумных протоколов имеет следующие ограничения:

- $N_R + N_W > N$
- $N_W > \frac{N}{2}$,

где N – общее количество серверов, N_R – кворум серверов готовых принять запрос на чтение, и N_W – кворум серверов на операцию обновления. Первое ограничение означает правило, при котором при чтении актуальных данных необходим как минимум один сервер из кворума записи, имеющий актуальные данные. Второе ограничение означает правило, при котором для записи обновленных данных необходим кворум из более, чем половины серверов.

5.2.1. Протоколы на базе алгоритма Paxos. Алгоритмы семейства Paxos призваны решить задачу консенсуса в сети ненадежных компонент [24]. Л. Лампорт предложил [43] базовый алгоритм Paxos, работающий с множеством входных данных и гарантирующий одно и только одно выходное значение. В алгоритме Paxos выделяются следующие роли:

- *Клиент*. Клиент отсылает запрос заявителю на обработку.
- *Заявитель*. Получает запросы от клиентов на обработку. Заявитель пытается получить кворум выборщиков для обработки данных, отправленных в запросе клиентом.

• *Выборщик*. Получает запросы от заявителя, одобряет или отклоняет их. Хранит в себе данные клиентских запросов. Помимо этого, содержит специальный целочисленный идентификатор, увеличивающийся каждый раз при одобрении обработать клиентский запрос.

В общих чертах, базовый алгоритм Paxos состоит из двух фаз [23]:

1. *Подготовительная фаза*. На этой фазе заявитель инициирует голосование, в котором получает либо одобрение от выборщиков на обработку клиентского запроса, либо отказ. Заявитель отправляет всем выборщикам сообщение с некоторым целочисленным идентификатором. Каждый из выборщиков принимает сообщение и, если идентификатор сообщения более свежий, чем хранящийся у выборщика, выборщик обещает не принимать сообщения с идентификатором меньше, чем у принятого сообщения. После чего выборщик посылает положительный ответ заявителю.

2. *Фаза принятия*. Как только заявитель получает одобрение выборщиков, он отправляет данные всем выборщикам, согласным принять клиентский запрос.

Один из примеров системы, в которой используется Paxos, рассмотрен в работе [41]. Система организована как множество подмножеств серверов баз данных. В каждом из множеств выбирается локальный лидер. С помощью алгоритма двухфазной фиксации результат работы одного подмножества может быть распространен другим подмножествам серверов [34].

Еще одим примером использования алгоритма Paxos является система, разработанная Дж. Бэкером с коллегами под названием Megastore [42]. Данная система поддерживает ACID семантику, хорошо масштабируется. Paxos используется для синхронизации данных по всем серверам.

Помимо базового алгоритма Paxos, существуют его различные расширения. Одно из них – Multi-Paxos. В случае базового алгоритма Paxos запросы поступают от одного клиента. В реализации Multi-Paxos клиентов может быть более одного. В таком случае возможны ситуации, когда одновременно проходят несколько стадий голосования по разным клиентским запросам. Выполнение параллельных фаз голосования в распределенной базе данных может привести к тому, что порядок получившихся результатов голосования может привести все сервера в системе к несогласованному состоянию. Такая проблема может быть решена путем использования журнала серверов, описанного в работе [24]. В работе [5] представлен пример реализации алгоритма Multi-Paxos в связке с внешней моделью согласованности.

Стоит заметить, что в системе из $2F + 1$ серверов одновременно не могут отказать F серверов. Например, в системе из 100 серверов могут отказать 10. В таком случае для принятия решения необходим кворум из 21 сервера.

Немалый интерес представляют другие алгоритмы из семейства Paxos. Один из них – Cheap Paxos [44]. Алгоритм отличается от базового алгоритма Paxos количеством $F + 1$ работающих серверов. Другие F вспомогательных серверов необходимы для переконфигурации системы в случае отказа менее половины серверов. В базовой версии алгоритма требуется для работы $2F + 1$ серверов.

В отличие от систем, где используется любой консенсус-алгоритм, в системах, где используется 2PC для фиксации обновления на всех серверах, отказ хотя бы одного сервера останавливает работу всей системы. При обновлении данных на всех серверах через алгоритм двухфазной фиксации пропускная способность системы ниже, чем в системах на базе алгоритма консенсуса.

5.2.2. Протокол на базе алгоритма Raft. Алгоритм Raft рассматривается как альтернатива семейству алгоритмов Paxos. В отличие от Paxos, алгоритм Raft более понятен в ходе изучения и прост в реализации. Разработчики алгоритма утверждают, что по характеристикам отказоустойчивости и пропускной способности он ничем не уступает алгоритмам семейства Paxos [45, 46]. Исследователи формально доказали корректность данного алгоритма [45, 47].

Алгоритм Raft определяет следующие роли: *лидер*, *ведомый*, *кандидат*. Каждая из перечисленных ролей ответственна за определенные задачи. Raft разделяет две основные задачи:

• *Выборы лидера*. Изначально, все сервера в кластере выступают в роли кандидатов. Алгоритм выбора лидера начинается с фазы голосования. Каждый сервер рассылает сообщения, что готова стать лидером и голосует за других. В конечном счете один и только один сервер становится лидером. Дабы поддерживать статус лидера, сервер рассылает небольшие сообщения ведомым, подтверждая факт, что он все еще доступен и способен принимать клиентские запросы. В случае, если ведомые не принимают сообщения от лидера, они начинают заново выбирать лидера.

• *Репликация лога*. Лидер принимает запросы от клиента, подготавливает лог-сообщение, содержащее упорядоченное множество команд чтения и записи, которые должны быть выполнены на ведомых серверах, и отправляет сообщения ведомым. В случае, если лидер получает от большинства серверов готовность принять сообщение, лидер фиксирует результат локально, а затем рассылает сооб-

щения ведомым серверам из кворума. Каждая ведомая реплика фиксирует лог-сообщение локально.

Задача алгоритма Raft, как и алгоритма Paxos, – принятие решения о готовности обработать клиентский запрос. Основное отличие алгоритма Raft от Paxos состоит в том, что в Raft используется сервер-координатор, принимающий клиентские запросы. В Paxos обработать клиентский запрос может любой сервер. Другим отличием алгоритма Raft от Paxos является фаза голосования. Если в Raft отсылается сообщение о готовности принять новые данные, то в алгоритме Paxos, как уже было сказано, сервер с ролью выборщика сравнивает специальное целочисленное значение, принятое от заявителя с идентификатором, хранимым в выборщике. От сравнения идентификаторов зависит сможет ли сервер обработать клиентский запрос. Недостатком алгоритма Raft является задержка при передаче сообщений между серверами.

На данный момент известны десятки реализации алгоритма Raft [46]. Алгоритм Raft используется в распределенной базе данных CockroachDB [18].

5.3. Активная репликация

Согласно [33], основная идея протокола активной репликации заключается в том, что все сервера принимают и обрабатывают одни и те же клиентские процессы. Это существенное отличие от алгоритмов на основе консенсуса, в которых подмножество серверов обрабатывают клиентский процесс. Согласованность данных гарантируется тем, что все сервера получают на вход один и тот же порядок клиентских процессов и гарантировано возвращают одинаковые ответы на запросы. Клиентский процесс взаимодействует не с одним сервером, а со всеми. Достоинством данного подхода является его простота. Недостатком протокола активной репликации является сложность обработки ситуации, в которой необходимо восстановить данные одного из поврежденных серверов.

Активная репликация используется крайне редко. Это обусловлено тем, что операция обновления всех серверов достаточно дорогостоящая. Алгоритмы на основе кворумных протоколов выглядят предпочтительнее.

6. ДРУГИЕ ПОДХОДЫ К ЗАДАЧЕ РЕПЛИКАЦИИ

В данном разделе рассмотрены работы, использующие различные модели согласованности.

В работе [22] представлен обзор схем репликации и алгоритмов распространения данных по всем серверам. Исследователи не провели обзор классов моделей согласованности и их реализаций.

В динамической, распределенной, отказоустойчивой, масштабируемой NoSQL базе данных MongoDB [17] используется согласованность в конечном счете. Данная система предоставляет возможность выбора одного из двух режимов синхронизации реплик. Идея первого режима состоит в выделении основного сервера, принимающего запросы клиентов, и ведомых. Во втором режиме любой сервер может принимать клиентские запросы.

В работе [8] представлен новый репликационный протокол TAPIR, основаный на слабой модели согласованности. TAPIR предоставляет все свойства ACID семантики. Разработчики в [8] смогли добиться фиксации транзакции в распределенной системе за одну фазу. Также была реализована возможность децентрализованной коммуникации между серверами.

Google предложила высокопроизводительную систему BigTable [19]. Динамическая система использует согласованность в конечном счете. BigTable предоставляет возможность через настройки выбрать модель согласованности *читай то, что записал*.

Amazon разработала децентрализованную систему баз данных Dynamo [21]. Система поддерживает согласованность в конечном счете в связке с кворум-алгоритмом Paxos. Представленная база данных хорошо масштабируется; отказоустойчивость гарантируется выбором нового лидера-координатора в случае, если предыдущий лидер недоступен.

В статье [26] представлено распределенное решение, основанное на ситуационно-зависимой модели согласованности в распределенной базе данных. Используя ситуационно-зависимую модель согласованности и выполнения неблокирующих операции чтения ко всем серверам, COPS гарантированно возвращает клиенту согласованные данные.

В статье [48] представлен фреймворк для NoSQL решений. Решение реализует модель строгой согласованности и предоставляет возможность динамически масштабировать распределенную базу данных. Фреймворк использует алгоритм четырехфазной фиксации4PC, гарантирующий свойства ACID семантики. Недостатком системы является высокое время отклика вследствие обмена сообщениями между серверами.

7. ЗАКЛЮЧЕНИЕ

В работе рассматриваются классы моделей согласованности и протоколы синхронизации серверов. Репликация данных решает проблему отказоустойчиво-

сти в распределенной базе данных, повышает пропускную способность чтения данных. Также, были проанализированы различные работы, использующие описанные в статье модели согласованности.

СПИСОК ЛИТЕРАТУРЫ

1. *Andrew S.* Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. ISBN 0132392275.
2. *Cheung S.Y., Casavant T.L., Singhal M., Ahamad M., Ammar M.H.* Replicated data management in distributed systems. 1994.
3. *Jiménez-Peris R., Pati no-Martínez M., Kemme B., Alonso G.* How to select a replication protocol according to scalability, availability, and communication overhead. In *SRDS*, 2001.
4. *Anderson T., Breitbart Yu., Korth H.F., Wool A.* Replication, consistency, and practicality: Are these mutually exclusive? *SIGMOD Rec.* 1998. V. 270 (2). P. 484–495. ISSN 0163-5808.
5. *Corbett J.C., Dean J., Epstein M., Fikes A., Frost Ch., Furman J.J., Ghemawat S., Gubarev A., Heiser Ch., Hochschild P., Hsieh W., Kanthak S., Kogan E., Li H., Lloyd A., Melnik S., Mwaura D., Nagle D., Quinlan S., Rao R., Rolig L., Yasushi Saito, Szymaniak M., Taylor Ch., Wang R., Woodford D.* Spanner: Googles globally distributed database. *ACM Trans. Comput. Syst.* 2013. V. 310 (3). P. 8:1–8:22. ISSN 0734-2071.
6. *Mani Chandy K. and Lamport L.* Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.* 1985. V. 30 (1). P. 63–75. ISSN 0734-2071. https://doi.org/10.1145/214451.214456. URL http://doi.acm.org/10.1145/214451.214456.
7. *Bailis P., Ghodsi A.* Eventual consistency today: Limitations, extensions, and beyond. *Queue*. 2013. V. 110 (3). P. 20:20–20:32. ISSN 1542-7730.
8. *Zhang I., Sharma N.Kr., Szekeres A., Krishnamurthy A., Ports D.R.K.* Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP'15. New York, NY, USA. 2015. P. 263–278 ACM. ISBN 978-1-4503-3834-9.
9. *Oki B.M., Liskov B.H.* Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC'88, pages 8–17, New York, NY, USA, 1988. ACM. ISBN 0-89791-277-2.
10. *Mansouri N., Dastghaibyfard G.H., Mansouri E.* Combination of data replication and scheduling algorithm for improving data availability in data grids. *Journal of Network and Computer Applications*. 2013. V. 360 (2). P. 711–722. ISSN 1084-8045.
11. *Elmirghani J.M.H., El-Gorashi T.E.H.* Data replication schemes for a distributed storage scenario. Munich, Germany, 07 2010. IEEE Computer Society. ISBN 978-1-4244-7798-2.
12. *UroEЎ ДЊibej, BoEЎtjan Slivnik, and Borut RobiДЌ.* The complexity of static data replication in data grids. *Parallel Computing*, 2005. V. 310 (8). P. 900–912, 2005. ISSN 0167-8191.
13. *Chervenak A., Deelman E., Foster I., Guy L., Hoschek W., Iamnitchi A., Kesselman C., Kunszt P., Ripeanu M., Schwartzkopf V. et al.* Giggle: A framework for constructing scalable replica location services. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, page 1–17. IEEE Computer Society Press, IEEE Computer Society Press, 2002.
14. *CitusDB official website*, 2019. URL www.citusdata.com.
15. *PostgreSQL official website*, 2019. URL www.postgresql.org.
16. *MySQL official website*, 2019. URL www.mysql.com.
17. *Banker K. Mongo DB in Action*. Manning Publications Co., Greenwich, CT, USA, 2011. ISBN 1935182870, 9781935182870.
18. *Cockroach DB official website*, 2019. URL www.cockroachlabs.com.
19. *Chang F., Dean J., Ghemawat S., Hsieh W.C., Wallach D.A., Burrows M., Chandra T., Fikes A., Gruber R.E.* Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.* 2008. V. 260 (2). P. 4:1–4:26. ISSN 0734-2071.
20. Lakshman A., Malik P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.* 2010. V. 440 (2). P. 35–40. ISSN 0163-5980.
21. *DeCandia G., Hastorun D., Jampani M., Kakulapati G., Lakshman A., Pilchin A., Sivasubramanian S., Vosshall P., Vogels W.* Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.* 2007. V. 410 (6). P. 205–220. ISSN 0163-5980.
22. *Martins V., Pacitti E., Valduriez P.* Survey of data replication in P2P systems. Research Report RR-6083, INRIA, 2006.
23. *Lamport L.* Paxos made simple. *ACM SIGACT News 32.*
24. *Van Renesse R., Altinbuken D.* Paxos made moderately complex. *ACM Comput. Surv.* 2015. V. 470 (3). P. 42:1–42:36. ISSN 0360-0300.
25. *Kenneth Gifford D. Information Storage in a Decentralized Computer System*. PhD thesis, Stanford, CA, USA, 1981. AAI8124072.
26. *Lloyd W., Freedman M.J., Kaminsky M., Andersen D.G.* Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP'11. New York, NY, USA, 2011. P. 401–416. ACM. ISBN 978-1-4503-0977-6.
27. *Zhu Y., Wang J.* Client-centric consistency formalization and verification for system with large-scale distrib-

uted data storage. *Future Generation Computer Systems*. 2010. V. 260 (8). P. 1180–1188, 2010. ISSN 0167-739X.

28. *Lamport L.* Readings in computer architecture. chapter How to Make a Mulitprocessor Computer That Correctly Executes Multiprocess Programs, pages 574–575. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. ISBN 1-55860-539-8.
29. *Petersen K., Spreitzer M.J., Terry D.B., Theimer M.M., Demers A.J.* Flexible update propagation for weakly consistent replication. *SIGOPS Oper. Syst. Rev.* 1997. V. 310 (5). P. 288–301. ISSN 0163-5980.
30. *Petersen K., Spreitzer M.J., Theimer M.M., Welch B.B., Terry D.B., Demers A.J.* Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, Austin, TX, USA, 1994. IEEE. ISBN 0-8186-6400-2.
31. *Bernstein P.A., Das S.* Rethinking eventual consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD'13, pages 923–928, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2037-5. URL http://dl.acm.org/10.1145/2463676.2465339.
32. *Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems.* O'Reilly, 2016. ISBN 978-1-4493-7332-0.
33. *Wiesmann M., Pedone F., Schiper A., Kemme B., Alonso G.* Understanding replication in databases and distributed systems. In *Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, ICDCS'00. Washington, DC, USA, 200. P. 464. IEEE Computer Society. ISBN 0-7695-0601-1.
34. *Daniel A. and Tatuo Nakanishi*. Performance evaluation of a two-phase commit based protocol for ddbs. ACM, 1982. P. 247–255. ISBN 0-89791-070-2.
35. *Gray J., Helland P., O'Neil P., Shasha D.* The dangers of replication and a solution. *SIGMOD Rec.*, 250 (2):0 173–182, June 1996. ISSN 0163-5808.
36. *Budhiraja N., Marzullo K., Schneider F.B., Toueg S.* Distributed systems (2nd ed.). chapter The Primary-backup Approach, pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993. ISBN 0-201-62427-3.
37. *Streaming replication*, 2019. URL postgrespro.ru/docs/postgrespro/11/warm-standby.
38. *Oracle official website*, 2019. URL www.oracle.com.
39. *Rane D. and Mahendra Dhore*. Overview of data replication strategies in various mobile environment. 04 2016.
40. *Thomas R.H.* A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.* 1979. V. 40 (2). P. 180–209.
41. *Glendenning L., Beschastnikh I., Krishnamurthy A., Anderson T.* Scalable consistency in scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP'11. New York, NY, USA, 2011. P. 15–28. ACM. ISBN 978-1-4503-0977-6.
42. *Baker J., Bond C., Corbett J.C., Furman J.J., Khorlin S., Larson J., Leon J.-M., Li Y., Lloyd A., Yushprakh V.* Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.
43. *Lamport L.* The part-time parliament. *ACM Trans. Comput. Syst.* 1998. V. 160 (2). P. 133–169. ISSN 0734-2071.
44. *Lamport L., Massa M.* Cheap paxos. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN'04. Washington, DC, USA, 2004. P. 307. IEEE Computer Society. ISBN 0-7695-2052-9.
45. *Ongaro D., Ousterhout J.* In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14. Berkeley, CA, USA, 2014. P. 305–320. USENIX Association. ISBN 978-1-931971-10-2.
46. *Raft consensus algorithm website*, 2019. URL https://raft.github.io/.
47. *Woos D., Wilcox J.R., Anton S., Tatlock Z., Ernst M.D., Anderson T.* Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016. P. 154–165, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4127-1.
48. *Lotfy A.E., Saleh A.I., El-Ghareeb H.A., Ali H.A.* A middle layer solution to support acid properties for nosql databases. *Journal of King Saud University – Computer and Information Sciences*. 2016. V. 280 (1). P. 133–145, 2016. ISSN 1319-1578.

ПРОГРАММИРОВАНИЕ, 2020, № 5, с. 72

ПОПРАВКА

НЕКОТОРЫЕ НЕДОСТАТКИ ВХОДНОГО СИНТАКСИСА KEYMAERA

© 2020 г. Т. Баар

DOI: 10.31857/S013234742005009X

В номер 4, 2020 г. вносится следующее изменение: в содержании номера меняется название статьи “Некоторые недостатки входного синтаксиса KeYmaera” автора Т. Баара. Англоязычное название статьи “A Metamodel-based Approach for Adding Modularization to KeYmaera’s Input Syntax” меняется на “Some Deficiencies of the KeYmaera’s Input Syntax”.

“Some Deficiencies of the KeYmaera’s Input Syntax” является дословным переводом названия с русского языка. Изменение вызвано снятием с публикации статьи из англоязычной версии журнала.