МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(ФГБОУ ВО «ВГУ»)

Факультет прикладной математики, информатики и механики

Кафедра математического обеспечения ЭВМ

Клиент-серверное приложение для формальной верификации программ

Бакалаврская работа
Направление 01.03.02 Прикладная математика и информатика
Профиль Системное программирование и компьютерные технологии

Допущено к защите в ГЭК 15.06.2016

Зав. кафедрой	_____ (подпись)	д. ф-м. н., доцент Махортов С.Д.
Обучающийся	_____ (подпись)	Ерин А.В.
Руководитель	_____ (подпись)	д. ф-м. н., доцент Махортов С.Д.

Воронеж 2016

АННОТАЦИЯ

Целью данной работы является построение клиент-серверной системы, позволяющей проводить формальную верификацию программ. Формальная верификация является одним из методов установления соответствия программы ее спецификации. Она является более надежной, чем тестирование и представляет собой менее трудоемкий и более автоматизированный процесс, чем формальное доказательство.

Объектом изучения служит один из методов формальной верификации – проверка моделей (model checking). Он кажется наиболее интересным ввиду простоты использования и возможности полностью автоматизировать процесс верификации, требуя от пользователя только текст программы и спецификацию, заданную на языке темпоральной логики. В качестве предмета изучения выступают алгоритм символьной проверки на моделях, использующий двоичные решающие диаграммы (BDD), в большинстве случаев позволяющий избежать проблемы «комбинаторного взрыва», алгоритмы выполнения простейших логических и предикаторных операций над BDD, а также технологии построения клиент-серверных приложений (ориентированных на Web и платформу Android).

В результате было создано клиент-серверное приложение, позволяющее проверять выполняемость свойств спецификации, выраженных в логике деревьев вычислений (CTL), и начата работа над написанием библиотеки для работы с двоичными решающими диаграммами. Для задания алгоритма пользователем используется компактный C-подобный язык. Приложение достаточно легко расширяемо как в сторону усложнения языка для задания алгоритма, так и в сторону предоставления возможности выражать свойство на других типах темпоральных логик (например, LTL). В данной работе рассматриваются архитектура созданного приложения, использованные алгоритмы и технологии. Приложение применимо в качестве инструмента проверки соответствия программы ее спецификации и может использоваться любыми разработчиками программного обеспечения. Созданы как Web-клиент приложения, так и Android-клиент. Данная работа разделена на четыре основных раздела, текст ее занимает 70 страниц, включает в себя 56 иллюстраций, 3 таблицы; к ней прикреплены 7 приложений, работа опирается на 15 источников.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Разработка программного обеспечения – сложный и трудоемкий процесс. Однако часто не менее сложной задачей является установления того, что программа делает именно то, что указано в ее спецификации. Наиболее распространенным способом решения этой проблемы является тестирование, которое, как метко подметил Эдсгер Вибе Дейкстра, «… может весьма эффективно продемонстрировать наличие ошибок, но безнадежно неадекватно для демонстрации их отсутствия» [1]. Тем не менее, тестирование является достаточно надежным и, главное, относительно простым. Другим способом установления корректности поведения программы является формальное доказательство, базирующееся на системах автоматизированного доказательства теорем. Этот метод является наиболее точным и надежным. Основной минус этого подхода в том, что он является полуавтоматическим, требуя работы эксперта. Наконец, существует и третий способ проверки того, что программа удовлетворяет спецификации – формальная верификация. Рассмотрим один из подходов к формальной верификации – проверку моделей (model checking).

Определение 1. *Проверка моделей – метод автоматической формальной верификации параллельных систем с конечным числом состояний* [2].

Из самого названия этого метода следует, что проверяется не сама программа непосредственно, а некая модель, построенная на ее основе. Такой моделью служит модель Крипке $M = (S_0, S, R, L)$, где S – множество состояний программы, $S_0 \subseteq S$ – множество начальных ее состояний, $R \subseteq S \times S$ – множество переходов между состояниями, $L: S \to 2^{AP}$ – функция пометок из множества состояний во множество всевозможных подмножеств атомарных предикатов – логических условий, так или иначе относящихся к проверяемой программе [2]. Следует отметить, что процесс трансляции программы в структуру Крипке может быть автоматизирован почти полностью, требуя от пользователя лишь задания атомарных предикатов. Спецификация же программы представляется формулой темпоральной логики (в большинстве случаев это формула логики CTL или LTL – логики ветвящегося времени или линейного). Темпоральные логики учитывают то, как события протекают во времени, таким образом, привнося в обычную бинарную логику возможность описания взаимосвязи явлений по временной шкале. Темпоральная логика является мощным аппаратом, и с ее помощью можно задать спецификацию программы достаточно точно.

Таким образом, model checking является методом, хорошо «приближающим» программу и ее спецификацию, не требуя при этом каких-либо экспертных знаний у верификатора (возможно, лишь за исключением знания используемой в средстве верификации темпоральной логики; однако, темпоральные

логики просты в изучении). А значит, model checking является едва ли не самым предпочтительным средством проверки соответствия программы своей спецификации, сочетая в себе надежность формального доказательства и относительную простоту тестирования.

Основной проблемой метода проверки моделей является так называемая проблема «комбинаторного взрыва» [2]. Состояние структуры Крипке является «мгновенным снимком» значений переменных программы. Добавление одной переменной в программу может привести к увеличению числа состояний во столько раз, сколько существует различных значений добавленной переменной. Поэтому необходимо при разработке средств верификации пользоваться таким способом хранения структуры Крипке, который бы хотя бы частично решал эту проблему.

Актуальной является задача создания такого средства верификации, которое бы использовало вышеупомянутую технику хранения структуру Крипке и при этом не требовало от пользователя ее прямого задания; то есть, смогло бы сгенерировать структуру Крипке по имеющейся программе на формальном языке. Кроме того, не менее актуальным является организация подобного средства в соответствии с клиент-серверной архитектурой, ориентированной на Web и мобильные платформы, ведь практически ни одно из известных программных средств проверки моделей не поддерживает подобный подход [3]. А значит, можно выделить комплекс подзадач:

а) создание приложения-верификатора, реализующего технику model checking, что включает в себя:
    1) создание транслятора, переводящего программу на формальном языке в структуру Крипке;
    2) создание подпрограммы, строящей по темпоральной формуле некоторое ее представление, аналогичное представлению структуры Крипке;
    3) создание верификатора – модуля, который проверяет, верна ли темпоральная формула на структуре Крипке;
    4) поиск и использование такого алгоритма построения вышеназванных объектов, который хотя бы частично решал проблему «комбинаторного взрыва»;
б) создание клиент-серверного приложения, в качестве одного из модулей включающее приложение-верификатор, описанное в предыдущем пункте.

1. АЛГОРИТМ ПРОВЕРКИ МОДЕЛЕЙ

1.1. Введение в темпоральную логику. Как было отмечено во введении, темпоральная логика представляет собой расширение бинарной логики с введенным временем. В темпоральной логике время имеет относительный характер: мы не указываем, какой конкретно момент времени нас интересует, а лишь оперируем понятиями текущего, прошедшего и будущего времени, считая само время дискретным [4]. Итак, в дополнение к имеющимся логическим операторам вводится ряд временных операторов. Рассмотрим самые распространенные из них (здесь q – логическое высказывание):

а) унарный оператор F: Fq есть $true$ тогда и только тогда, когда q верно хотя бы раз в будущем;

б) унарный оператор G: Gq есть $true$ тогда и только тогда, когда q верно всегда в будущем;

в) унарный оператор X: Xq есть $true$ тогда и только тогда, когда q верно для следующего момента времени;

г) бинарный оператор U: pUq есть $true$ тогда и только тогда, когда p истинно вплоть до будущего момента времени, когда истинным становится q, при этом p может быть верным и далее [4].

Например, высказывание «когда-нибудь программа будет выполняться до тех пор, пока ее не остановят» можно записать как *FG(“программа выполняется” U “программу останавливают”)*.

Рассмотрим два наиболее часто встречающихся вида темпоральных логик. Первая из них – это логика линейного времени (LTL). Здесь, как следует из названия, время линейно и в каждый момент времени есть «отдельный мир», в котором LTL-формула либо истинна, либо ложна. При этом формула этой логики является формулой пути, то есть характеризует ряд состояний. Формально, формула LTL определяется следующим образом (здесь p – формула двоичной логики):

$$\varphi := p \mid \neg\varphi \mid \varphi \vee \varphi \mid X\varphi \mid \varphi\ U\ \varphi$$ [4].

При этом надо заметить, что $F\varphi = true\ U\ \varphi$, а $G\varphi = \neg F\neg\varphi$ [4].

Вторая логика – это логика ветвящегося времени (CTL). Также из названия понятно, что время здесь имеет ветвящуюся структуру, то есть для одного момента времени может существовать несколько следующих моментов. В CTL каждый темпоральный оператор снабжен квантором всеобщности A (означает требование верности формулы на всех путях, исходящих из вершины, соответствующей начальному моменту времени) или существования E (хотя бы на одном из путей) [4]. Формула этой логики является формулой состояния. Определим формально CTL-формулу:

$\varphi := p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid EF\varphi \mid EG\varphi \mid E(\varphi U\varphi) \mid AX\varphi \mid AF\varphi \mid AG\varphi \mid A(\varphi U\varphi)$ [4].

Необходимо отметить, что и LTL, и CTL являются подмножествами логики CTL* [2]. При этом существуют LTL-формулы, невыразимые в CTL (например, $FG\varphi$ – "стабилизация свойства φ когда-нибудь в будущем"), и формулы CTL, невыразимые в LTL (например, $AGEF\varphi$ – "достижимость свойства φ").

1.2. Структура Крипке. Как было отмечено во введении, структура Крипке является моделью для представления программы, включая в себя 4 компоненты: множество состояний S, множество стартовых состояний $S_0 \subseteq S$, отношение $R \subseteq S \times S$ переходов, функция $L: S \to 2^{AP}$ пометок из множества состояний во множество всевозможных подмножеств атомарных предикатов. Часто к структуре Крипке добавляют еще один компонент: множество атомарных предикатов AP. Рассмотрим каждый из компонентов подробнее.

Состояние фактически представляет собой «мгновенный снимок» значений переменных программы. Часто к программным переменным добавляют еще одну: *ProgramCounter* (*PC*), которая хранит номер выражения программы, в котором переменные принимают определенные значения. Приведем небольшой пример. Пусть есть фрагмент программы на неком C-подобном языке:

```
int a, b;
a = 5;
if (a > 3) {
   b = 10;
}
else {
   b = 15;
}
```

Разметим его, пронумеровав каждую значащую строку:

```
0: int a, b;
1: a = 5;
   if (a > 3) {
2:    b = 10;
   }
  else {
3:    b = 15;
  }
```

Значения PC как раз совпадают с номерами строк. Опишем множество состояний этого фрагмента программы: *S = {s0, s1, s2, s3}*, где *s0 = {pc = 0}, s1 = {pc = 1, a = 5}, s2 = {pc = 2, a = 5, b = 10}, s3 = {pc = 3, a = 5, b = 15}*. При этом множество S_0 стартовых состояний есть *{s0}*, а отношение переходов R = *{(s0, s1), (s1, s2), (s1, s3)}*.

Атомарные предикаты же есть формулы двоичной логики, относящиеся к спецификации программы. Например, для приведенного выше примера предикаты могут быть такими: $a < b$, $a + b != a * b$, $b > 16$ и так далее.

Функция $L: S \to 2^{AP}$ есть функция, которая каждому состоянию сопоставляет множество атомарных предикатов, которые выполняются в этом состоянии. В приведенном выше примере состояниям *s0* и *s1* не соответствует ни один предикат из предыдущего абзаца, *s2* и *s3* – первый и второй.

1.3. Выбор способа хранения структуры Крипке. Рассмотрим некоторые способы хранения структуры Крипке, отмечая их плюсы и минусы. Первый способом является наивный способ: непосредственное хранение всех компонентов структуры Крипке в структуре, подобной массиву одномерных массивов, где каждый одномерный массив представляет собой состояние из множества S, просто храня значения программных переменных. В этом случае отношение переходов R можно представить как массив, элементом которого является пара ссылок на состояния из S, а множество стартовых состояний S_0 - как набор ссылок на элементы S. Плюс такого подхода в простоте реализации, минус же очевиден – он подвержен «комбинаторному взрыву». Тем не менее, средство формальной верификации Java Path Finder (JPF) использует подобный алгоритм, накладывая некоторые ограничения на размер программы. Однако следует отметить, что JPF в основном служит для проверки программ на отсутствие проблем, связанных с использованием более чем одного потока в программе (например, взаимные блокировки).

Распространенными алгоритмами проверки моделей являются алгоритмы, использующие различные автоматы Бюхи. Они в большинстве случаев не подвержены «комбинаторному взрыву», однако работают лишь с логикой линейного времени (LTL) [2].

Другим интересным вариантом является способ хранения структуры Крипке, использующий упорядоченные двоичные разрешающие диаграммы (OBDD). OBDD позволяют в большинстве случаев решить проблему «комбинаторного взрыва» [2]. Более подробно об этом написано в параграфе 4 главы 1. Кроме того, библиотеки работы с OBDD существуют для многих языков программирования. OBDD поддерживают представления как CTL, так и LTL формул, а работа с OBDD достаточно легко поддается отладке [2].

Таким образом, было принято решение использовать OBDD для хранения структуры Крипке и представления темпоральной формулы.

1.4. Описание алгоритма model checking, основанного на OBDD. В этом параграфе дано общее представление о OBDD, а также описан алгоритм, который по имеющемуся программному коду строит OBDD соответствующей структуры Крипке и по имеющейся темпоральной формуле – OBDD, соответст-

вующую ей. В этой работе было принято решение ограничиться логикой ветвящегося времени (CTL). Об аналогичном алгоритме для LTL можно прочитать в [2].

1.4.1. Начальные сведения о OBDD. Пусть φ – булева функция трех переменных x, y и z, а ее таблица истинности выглядит так:

Таблица 1. Таблица истинности функции

x	y	z	Ф
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Построим графическое представление этой функции (рисунок 1):

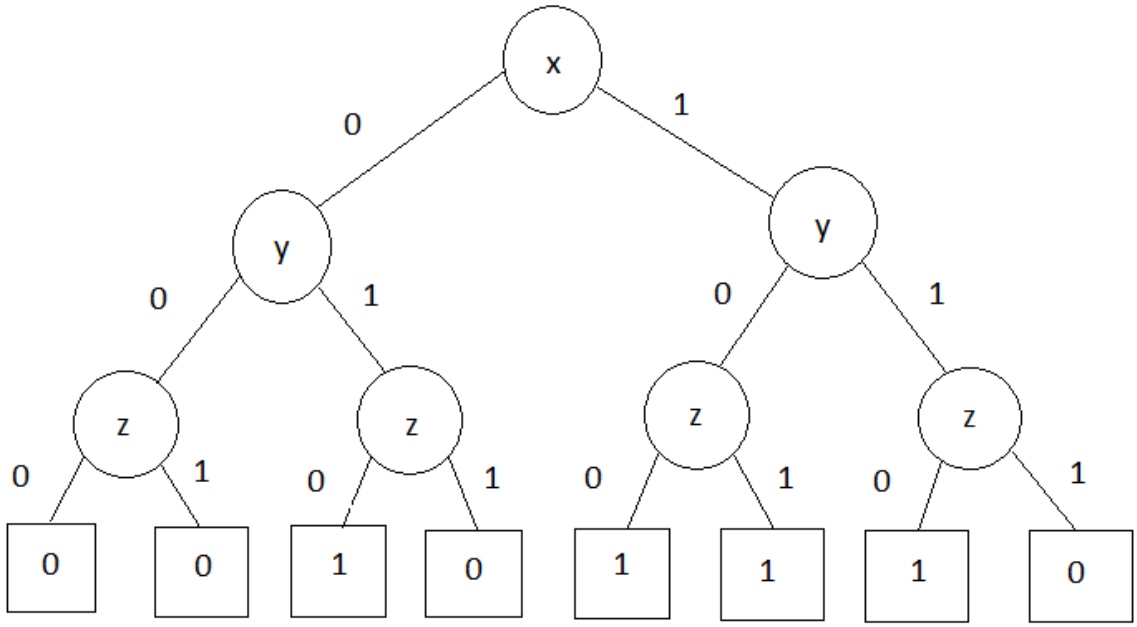

Рисунок 1. Графическое представление функции

Зафиксируем порядок переменных в этой функции: $x < y < z$. Далее, воспользуемся двумя правилами:

а) объединить любые изоморфные подграфы;

б) удалить из графа любую вершину, у которой оба непосредственных преемника изоморфны [2].

Будем применять их к графу, изображенному на рисунке 1, до тех пор, пока ни одна из этих операций не будет давать эффекта.

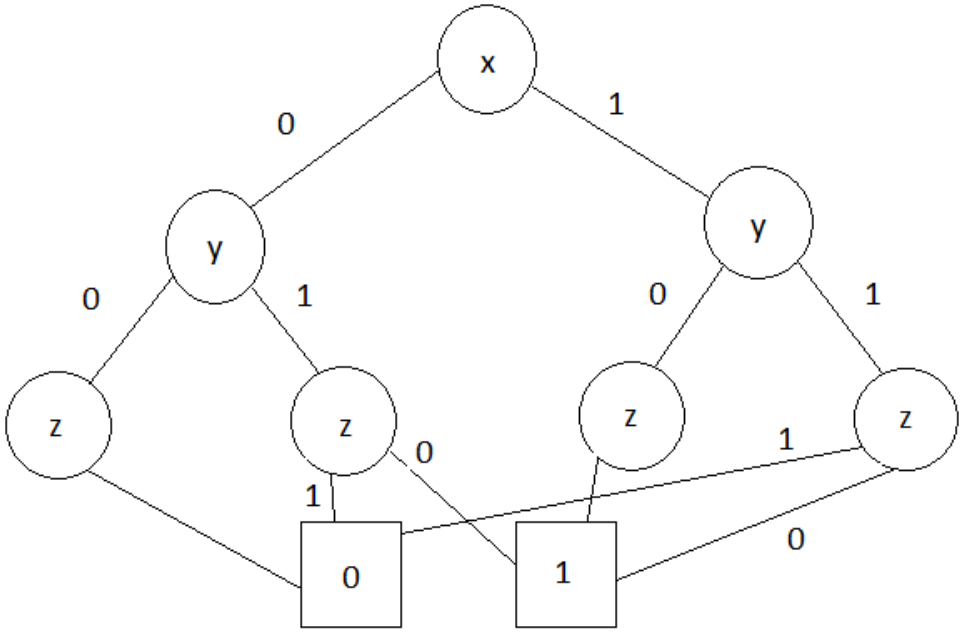

Рисунок 2. Объединение терминальных вершин

На рисунке 2 изображен граф, к которому применено правило а). Применим к нему правило б) (рисунок 3):

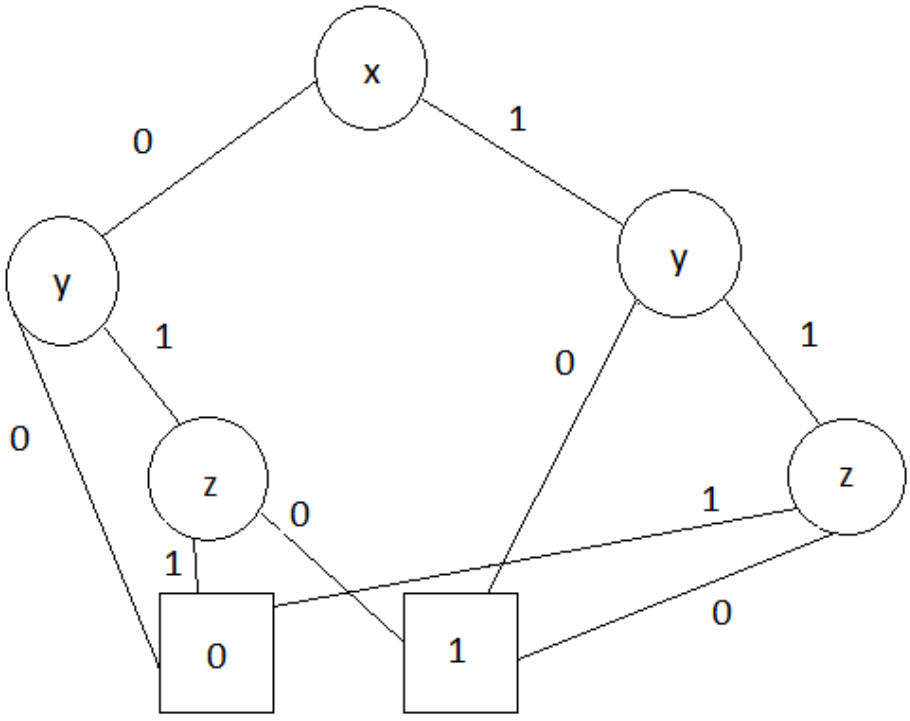

Рисунок 3. Удаление вершин

Видим, что при закрепленном порядке переменных дальнейшее применение правил а) и б) не приводит к изменению графа. Представление функции на рисунке 3 и есть ее OBDD.

Определение 2. *Двоичная упорядоченная разрешающая диаграмма (OBDD) – это двоичный направленный граф без избыточностей (редуцированный ациклический орграф) с одной корневой вершиной и в общем случае двумя листьями, помеченными 0 и 1.Нелистовые вершины помечены переменными, из каждой из них выходят ровно два ребра, соответствующие значениям 0 и 1 переменной, помечающей вершину* [4].

Теорема 1.*OBDD является каноническим представлением булевой функции при любом фиксированном порядке переменных* [2].

При одном и том же порядке переменных бинарные логические операции над OBDD оказываются достаточно простыми, а представление булевой функции в таком виде часто оказывается более компактным, чем СКНФ, СДНФ и полином Жегалкина [2]. Кроме того, OBDD является графом, и поэтому над ней осуществимы такие операции, как прямой и обратный образы.

При этом следует заметить, что размер и сложность построения OBDD зависит от выбранного порядка переменных в булевой функции.

1.4.2. Связь OBDD и характеристической функции подмножества конечного множества. Пусть M – некоторое конечное множество, $U \subseteq M$. Представим каждый элемент M уникальным набором единиц и нулей. Таким образом, каждый элемент множества M представляет собой набор значений некоторых булевых переменных. Пусть χ_U - характеристическая функция множества U. Тогда χ_U можно интерпретировать как булеву функцию, которая принимает значение 1 на тех и только на тех наборах значений булевых переменных, которые соответствуют элементам множества M, входящих в U. И тогда понятно, что для χ_U можно построить OBDD. Таким образом, OBDD может представлять подмножество конечного множества.

1.4.3. Представление структуры Крипке с помощью OBDD. Из предыдущего параграфа совершенно ясно, что структура Крипке может быть представлена как набор OBDD. Так, множество S состояний есть подмножество множества $\{0,1\}^n$, где n – минимальное число булевых переменных, нужных для кодирования состояний структуры Крипке. S_0– множество стартовых состояний – есть подмножество множества S, а R – множество переходов между состояниями – подмножество $S \times S$. Для каждого атомарного предиката можно хранить OBDD, представляющую подмножество S состояний, в которых верен данный предикат.

1.4.4. Трансляция формулы CTL в OBDD. Построим алгоритм представления CTL-формулы на имеющейся структуре Крипке в виде OBDD. CTL-формула, выражающая спецификацию программы, представляет собой применение темпоральных и логических операторов к атомарным предикатам. Заметим, что эта задача представляет собой задачу построения OBDD для характе-

ристической функции χ_φ заданной CTL-формулы φ на множестве состояний S структуры Крипке. Ясно, что обычные логические операции представляют собой аналогичные операции над OBDD, представляющими операнды. Рассмотрим непосредственно темпоральные операторы.

Оператор EX. Пусть R – OBDD, представляющая переходы структуры Крипке. Здесь и далее будем обозначать через φ некоторое выражение CTL, для которого известно его представление в виде OBDD (то есть OBDD, представляющая состояния структуры Крипке, в которых верно φ), а саму OBDD – через Φ. Результат EXφ есть все такие состояния структуры Крипке, из которых есть переход хотя бы в одно состояние, в котором верно φ [4]. Далее, пусть $\{x_1, \dots, x_n\}$ – набор булевых переменных, необходимых для кодирования состояний структуры Крипке S. Тогда $\{x_1, \dots, x_n, x_1', \dots, x_n'\}$ – набор переменных, необходимых для кодирования R. Очевидно, что Φ зависит от переменных $x_1, \dots, x_n$. Сделаем следующее: «сдвинем» Φ так, чтобы она зависела от переменных $x_1', \dots, x_n'$. Теперь пусть $\Psi = \Phi \cap R$. Ясно, что Ψ представляет те состояния, в которых верно φ и в которые есть переход. Теперь просто применим к Ψ операцию «обратный образ» и получим результат применения оператора EX к φ.

Оператор EF. Представим его применение к φ в следующем виде: $EF\varphi = \varphi \vee EX(EF\varphi)$ [4]. Воспользуемся следующими утверждениями.

Теорема 2. *Оператор EF является монотонным на любом конечном множестве U, то есть для любых P, Q: $P \subseteq Q \subseteq U$, верно $EF(P) \subseteq EF(Q)$* [4].

Определение 3. *Неподвижной точкой оператора τ на множестве U называется подмножество V множества U, такое что $\tau(V) = V$* [4].

Теорема 3 (Тарского). *У любого монотонного оператора τ на конечном множестве U существуют максимальная и минимальная неподвижные точки (обозначающиеся как $\nu U.\tau(U)$ и $\mu U.\tau(U)$). Максимальную точку получаем, если взять в качестве начального множества все множество U, а минимальную – если $\emptyset$* [4].

Теорема 4. *Искомое множество S' состояний структуры Крипке, в которых верна формула $EF\varphi$, есть наименьшая неподвижная точка оператора EF как оператора на множестве S состояний структуры Крипке* [4].

Отсюда следует алгоритм нахождения OBDD для состояний, удовлетворяющий формуле $EF\varphi$ при известной OBDD для состояний, в которых верно φ:

а) взять в качестве начального множества пустое множество – идет нахождение минимальной точки EF на множестве S состояний структуры Крипке (итак, изначально $EF\varphi = \emptyset$);

б) вычислить правую часть равенства $EF\varphi = \varphi \vee EX(EF\varphi)$;

в) повторять шаг б) до тех пор, пока вычисление правой части не будет приводить к добавлению новых состояний.

Оператор AX. Имеем: $AX\varphi = \neg EX\neg\varphi$ [2].

Операторы AF, EG, AG, EU, AU. Алгоритм вычисления множества состояний, удовлетворяющих этим операторам, по сути своей аналогичен вышеприведенному алгоритму для EF. Поэтому приведем здесь таблицу, показывающую, каким образом можно найти состояния из *S*, удовлетворяющие перечисленным операторам (напомним, что $\nu U.\tau(U)$ обозначает максимальную неподвижную точку оператора τ на множестве U, а $\mu U.\tau(U)$ – минимальную) (включим в эту таблицу и EF) [4]:

Таблица 2. Рекуррентное представление темпоральных операторов

Рекуррентная форма оператора	*Неподвижная точка*
$EF\varphi = \varphi \vee EX(EF\varphi)$	$EF\varphi = \mu V.\varphi \vee EX(V)$
$AF\varphi = \varphi \vee AX(AF\varphi)$	$AF\varphi = \mu V.\varphi \vee AX(V)$
$EG\varphi = \varphi \wedge EX(EG\varphi)$	$EG\varphi = \nu V.\varphi \wedge EG(V)$
$AG\varphi = \varphi \wedge AX(AG\varphi)$	$AG\varphi = \nu V.\varphi \wedge AG(V)$
$E(\varphi_1 U\varphi_2) = \varphi_2 \vee (\varphi_1 \wedge EX(E(\varphi_1 U\varphi_2)))$	$E(\varphi_1 U\varphi_2) = \mu V.\varphi_2 \vee (\varphi_1 \wedge EX(V))$
$A(\varphi_1 U\varphi_2) = \varphi_2 \vee (\varphi_1 \wedge AX(A(\varphi_1 U\varphi_2)))$	$A(\varphi_1 U\varphi_2) = \mu V.\varphi_2 \vee (\varphi_1 \wedge AX(V))$

1.4.5. Алгоритм трансляции в OBDD программы без явного построения структуры Крипке. В этом пункте будет рассмотрено, каким образом программу, написанную на формальном языке, можно представить в виде OBDD, минуя явное предварительное построение структуры Крипке. Ограничимся небольшим C-подобным языком, имеющим арифметические и логические операторы, оператор присваивания, считывания (read), конструкции if-then, if-then-else, while (цикл с предусловием).

Пусть S_0 – OBDD, представляющая начальные состояния структуры Крипке, R – отношение переходов между состояниями, AP_i – состояния, в которых верен *i*-ый предикат. Так как в данной работе мы ограничиваемся CTL, то нам не нужно хранить состояния структуры Крипке, так как формула CTL выполнена на структуре Крипке, если она выполнена на ее стартовых состояниях [2]. Изначально все эти OBDD представляют тождественно равную нулю булеву функцию.

Предположим, что нам дана некоторая программа. Разметим, начиная с 0, ее, введя новую переменную *ProgramCounter* (*PC*). Стартовые состояния программы описываются лишь одним равенством: *PC = 0*. Далее, укажем, как транслировать вышеназванные программные конструкции в предикаты, характеризующие переходы. Здесь мы исходим из того, что все переменные известны

к моменту начала трансляции. Будем обозначать всё множество переменных как V, а запись $same(U), U \subseteq V$ будет означать, что все переменные из U не изменились. Через штрихованные переменные будем обозначать те, в которые выполняется переход.

а) *Оператор присваивания*. В размеченном виде он выглядит как *m1: a = A; m2:*. Переход из состояния *m1* в *m2* будет описываться следующим образом: $(PC = m1) \wedge (PC' = m2) \wedge (a' = A) \wedge same(V \backslash \{a\})$.

б) *Конструкция if-then-else*. Имеем: *m: if (b) { m1: P1 } else { m2: P2 } m3:*. Здесь получаем (*P1, P2* – набор некоторых выражений языка): $b \wedge (PC = m) \wedge (PC' = m1) \wedge same(V) \vee \neg b \wedge (PC = m) \wedge (PC' = m2) \wedge same(V) \vee$ {трансляция конструкций внутри *then*-блока с конечным переходом в *m3*} $\vee$ {трансляция конструкций внутри *else*-блока с конечным переходом в *m3*}. Ясно, что конструкция *if-then* получается из этой конструкции отбрасыванием частей, связанных с выполнением *else*-блока.

в) *Конструкция while*. Имеем: *m: while (b) { m1: P1 } m2:*. Получаем: $b \wedge (PC = m) \wedge (PC' = m1) \wedge same(V) \vee \neg b \wedge (PC = m) \wedge (PC' = m2) \wedge same(V) \vee$ {трансляция конструкций внутри *while*-блока с переходом либо в *m*, либо в *m2*}.

г) *Функция read считывания значения переменной*. У этого выражения есть одна особенность: если до него считываемая переменная не была инициализирована и переходы фактически не зависели от значения этой переменной, то после считывания значение этой переменной становится детерминировано. В остальном трансляция *read* ничем не отличается от трансляции оператора присваивания.

На каждом шаге трансляции к OBDD R, представляющей переходы между состояниями, с помощью дизъюнкции добавляется OBDD или набор OBDD, представляющие текущий переход.

Далее, на каждом шаге трансляции проверяется, удовлетворяет ли текущее состояние каждому из атомарных предикатов. Если текущее состояние удовлетворяет *i*-му предикату, то оно также с помощью дизъюнкции добавляется к AP_i. Проблема построения такого транслятора является проблемой построения компилятора первого рода, переводящего программу в логическую формулу [4].

После этого происходит трансляция формулы CTL в OBDD: OBDD каждого атомарного предиката известна, и мы выполняем трансляцию логических и темпоральных операторов (см. п. 1.4.4).

1.4.6. Алгоритм верификации. Итак, пусть, как и в предыдущем пункте, S_0 – OBDD, представляющая начальные состояния структуры Крипке. Пусть C – результат трансляции формулы CTL, выражающей спецификацию данной программы, в OBDD. Формула CTL выполняется на структуре Крипке тогда и только тогда, когда она верна в стартовом состоянии структуры Крипке [2]. Отсюда получаем алгоритм верификации: найти OBDD дизъюнкции S_0 и C. Если эта OBDD равна C, то программа удовлетворяет своей спецификации; если же не равна, то программа не удовлетворяет своей спецификации [2]. Контрпримеры, показывающие, в каких стартовых состояниях неверна заданная формула CTL, можно получить, выполнив сложение по модулю два C и дизъюнкции S_0 и C.

1.5. Описание основных алгоритмов работы с OBDD. Приведем описание основных алгоритмов работы с OBDD: логических и предикаторных. Однако сначала рассмотрим, какая структура хранения более всего подходит для OBDD.

1.5.1. Структура хранения OBDD. Как было сказано в пункте 1.4.1, двоичная решающая диаграмма представляет собой граф. Заметим, что ни одна из вершин – за исключением терминальных – не может иметь более одного родителя. Также и любая вершина, не являющаяся терминальной, имеет двух и только двух потомков. В связи с этим напрашивается представление OBDD в виде следующей таблицы [5]:

Таблица 3. Структура хранения OBDD

Порядковый номер	Переменная	Родитель	0-направление	1-направление
0	undefined	[2, 3, …]	undefined	undefined
1	undefined	[2, 4, …]	undefined	undefined
2	x_0	3	0	1
…	…	…	…	…
n-1	x_{k-1}	n	n-3	n-6
n	x_k	undefined	n-2	n-1

Здесь столбец «переменная» отвечает за то, какой переменной в булевой функции соответствует вершина графа; «родитель» определяет то множество вершин, из которых в вершину идут дуги; «0-направление» – ту вершину, в которую из рассматриваемой вершины выходит дуга, помеченная нулем; «1-направление» – ту вершину, в которую из рассматриваемой вершины выходит дуга, помеченная единицей. Отметим, что столбец «родитель» может оказаться лишним в зависимости от реализации модуля OBDD.

1.5.2. Алгоритм интерпретации функции. Рассмотрим алгоритм интерпретации булевой функции, представленной в виде OBDD – алгоритм, который по данной OBDD строит OBDD, представляющую булеву функцию, в которой

вместо указанной переменной подставлено некоторое ее значение – логический нуль или логическая единица. Этот алгоритм является базовым алгоритмом для реализации логических операций булевой алгебры над OBDD.

Итак, пусть нам даны: OBDD f, переменная x и ее значение из множества $\{0, 1\}$ a. Также будем считать, что f представлена в виде описанной в предыдущем пункте таблицы. Сформируем новую OBDD, которая будет представлять собой булеву функцию, где над f выполнена операция интерпретации переменной x значением a. Обозначим вновь созданную OBDD через g. Отметим, что, если f не зависит от x, то g есть f. Будем далее считать, что f зависит от x. Ясно, что первым шагом получения g из f является исключение из последней всех вершин, в которые входят пути из всякой вершины, отвечающей переменной x, причем пути только такие, что первой дугой их является дуга из вершины, отвечающей x, помеченная как $\overline{a}$. При этом необходимо отметить, что терминальные вершины не исключаются из f, если после удаления всех описанных вершин на них ссылается хотя бы одна вершина. После первой итерации наступает вторая: из f удаляются все вершины, соответствующие переменной x (обозначим это множество вершин через X). Те вершины, которые остались после этого шага без родительской (очевидно, это те вершины, на которые ссылались элементы множества X дугами с весом a), становятся дочерними соответствующих вершин из множества родительских вершин для элементов X – и мы получаем искомую OBDD g.

Следует отметить, что этот алгоритм будет реализован эффективно с помощью описанной в предыдущем пункте структуры хранения, лишь если в ней будет возможен поиск строк по идентификатору переменной.

1.5.3. Логические алгоритмы. Рассмотрим алгоритмы реализации основных операций булевой алгебры над OBDD. Разделим этот пункт на две основные части: осуществление унарной операции отрицания и бинарных операций (конъюнкции, дизъюнкции, сложения по модулю два) над OBDD. Начнем с операции отрицания.

Условимся, что в таблице, приведенной в пункте 1.5.1, под ключом «0» всегда хранится запись, отвечающая терминальной вершине OBDD, помеченной нулем, под ключом «1» – вершине, помеченной единицей. Отметим, что если, например, конечной вершиной пути в OBDD является вершина «0», то функция, представляемая OBDD, принимает значение 0 на значениях, совпадающих с весом дуг пути, тех переменных, за которые отвечают вершины, входящие в этот путь. Таким образом, становится очевидно, что если перенаправить дуги, входящие в вершину, помеченную нулем, в вершину, помеченную единицей – и наоборот – то для данной функции f получим функцию $\overline{f}$. Необходимо отметить простоту и быстродействие этого алгоритма.

Теперь перейдем к операциям бинарным. Любая бинарная операция над булевыми функциями может быть осуществлена достаточно просто. Пусть $\odot$ – произвольная бинарная операция над булевыми функциями f и g. Имеем: $f \odot g = if(x = 0)\ then\ (f|_{x=0} \odot g|_{x=0}) else (f|_{x=1} \odot g|_{x=1})$ [6]. Необходимо отметить, что здесь x последовательно пробегает все переменные, от которых зависят f и g, начиная с той переменной, которая является первой в фиксированном для этих функций порядке переменных. Таким образом, мы приходим к следующему ограничению: *бинарные операции могут быть осуществимы эффективно над OBDD f и g, только если f и g имеют одинаковый порядок переменных; в противном случае понадобится переупорядочивание одной из OBDD.* Рассмотрим этот алгоритм применительно к OBDD на примере.

Пусть f – OBDD, представляющая функцию $F(y) = y$, g – OBDD, представляющая функцию $G(x, z) = x \vee z$, при этом переменные x, y, z упорядочены как $x < y < z$. Найдем $h = f \wedge g$.

Очевидно, что f имеет следующий вид (рисунок 4):

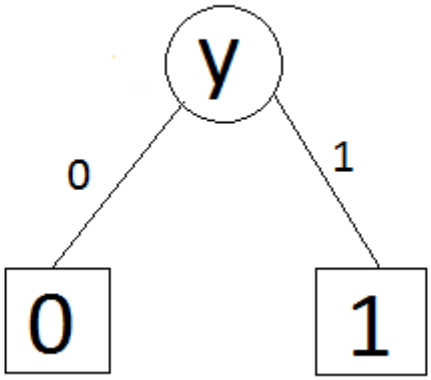

Рисунок 4. OBDD функции *f*

Также понятно, что g имеет следующий вид (рисунок 5):

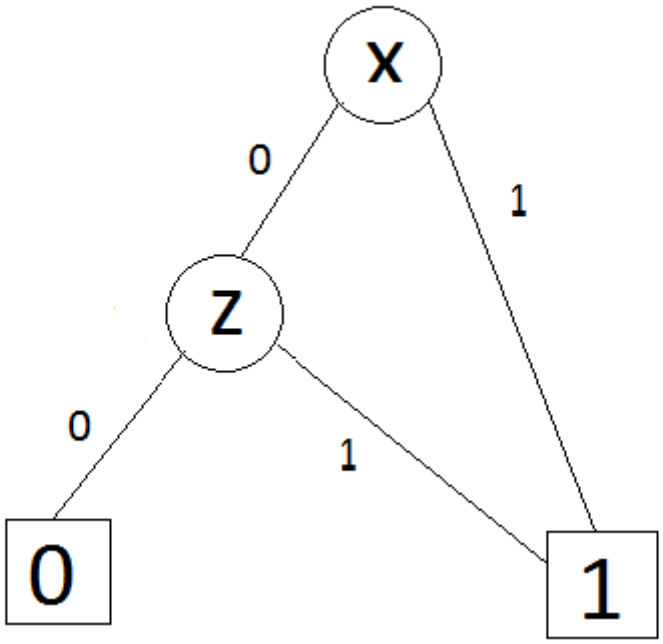

Рисунок 5. OBDD функции *g*

Будем применять последовательно написанное выше правило. Минимальный порядок среди трех переменных, от которых зависят f и g, имеет x. Первый шаг будет выглядеть так (рисунок 6):

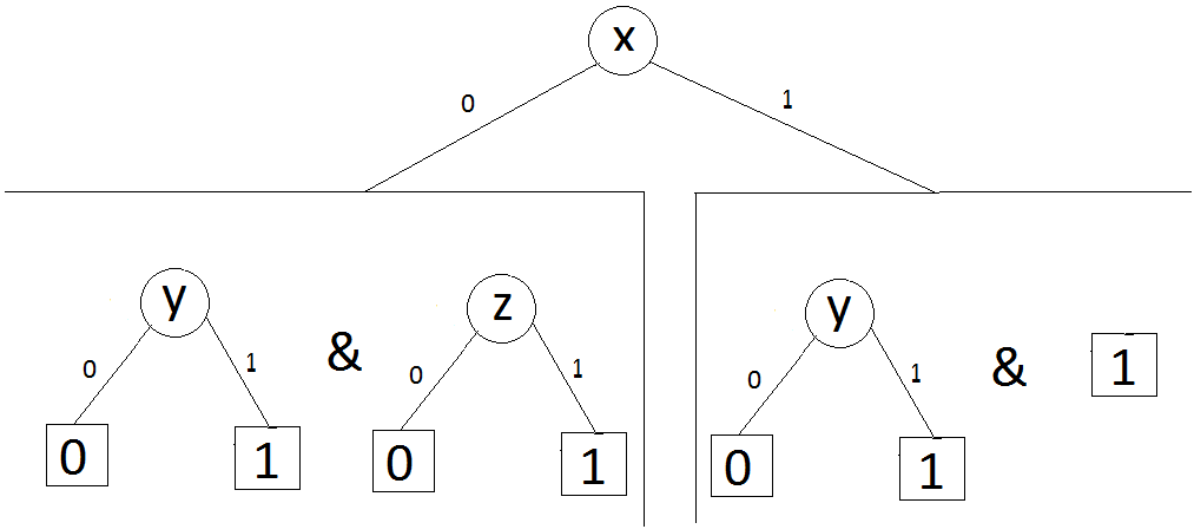

Рисунок 6. Разбиение по первой переменной

Видим, что OBDD, находящаяся в правой части, есть представление функции $U(y) = y$. Переходим ко второму шагу – конкретизации переменной y (рисунок 7):

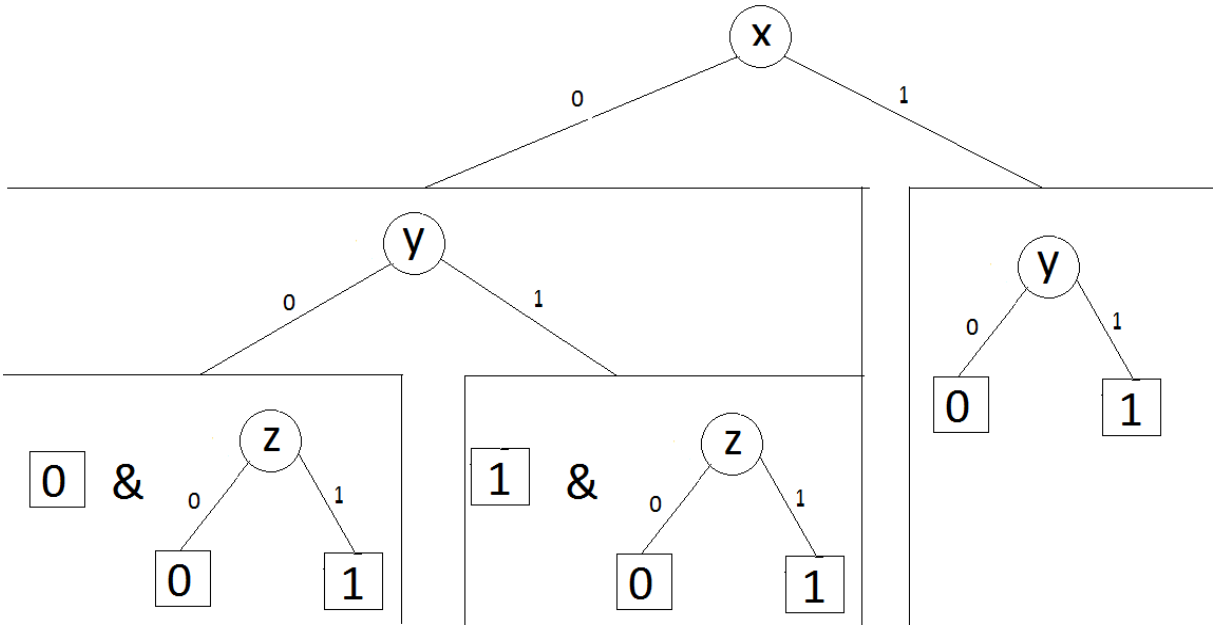

Рисунок 7. Разбиение по второй переменной

Видим, что более в конкретизации мы не нуждаемся. Поэтому начинаем слияние (рисунок 8):

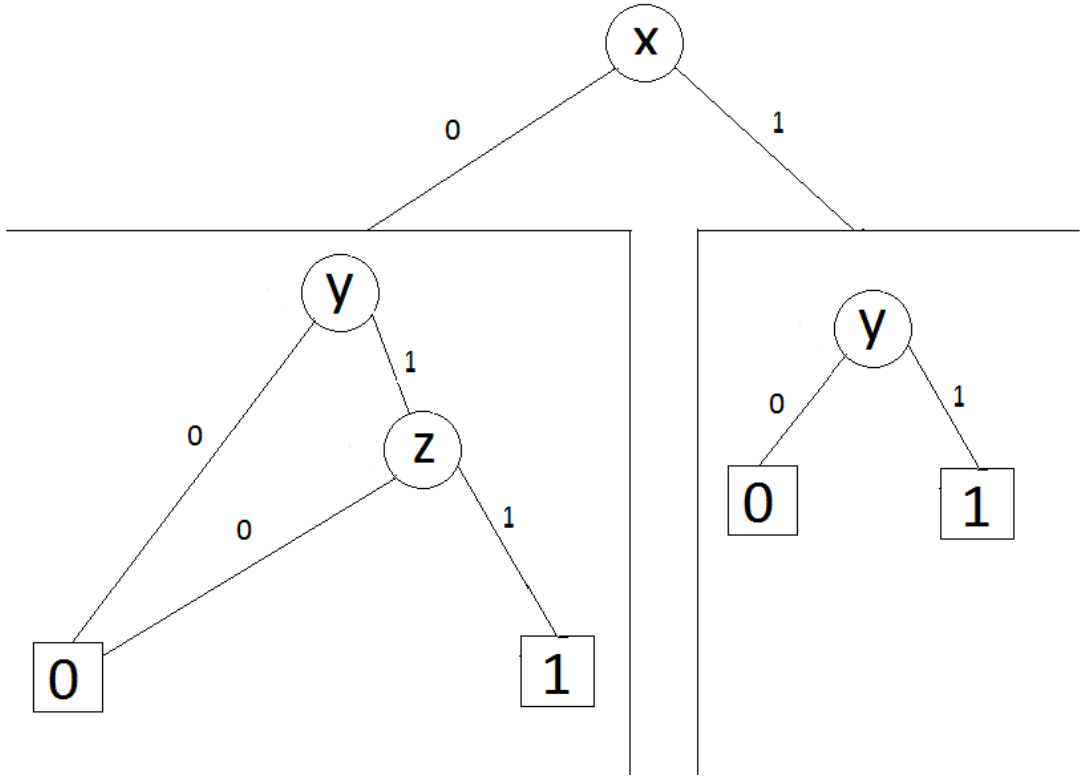

Рисунок 8. Слияние на уровне второй переменной

На каждом шаге слияния проверяем две сливаемых OBDD на равенство. Если две сливаемых OBDD равны, значит, достаточно переместить на следующий уровень любую из них.

Наконец, выполним последнее слияние (рисунок 9):

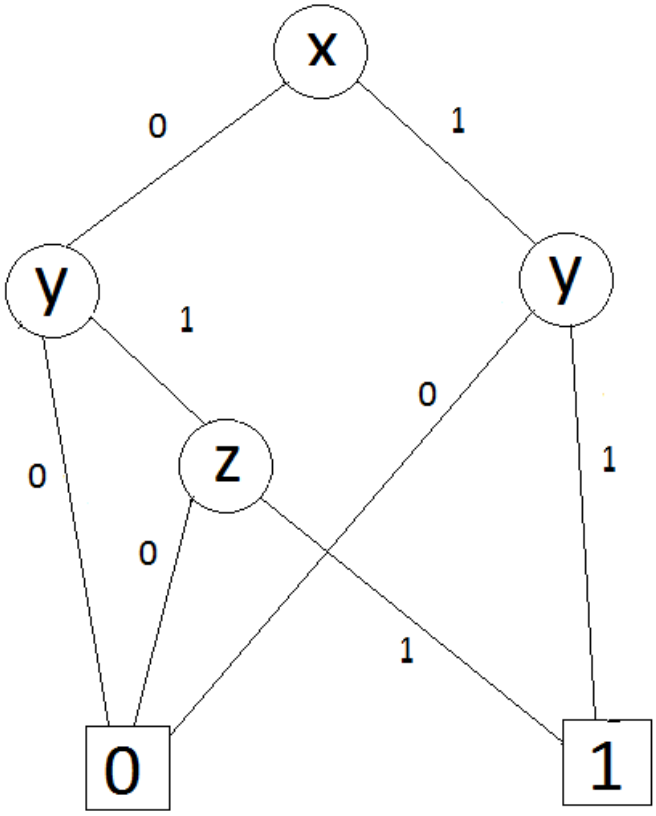

Рисунок 9. Слияние на уровне первой переменной

В итоге мы получили двоичную решающую диаграмму $f \wedge g$, представляющую булеву функцию $F \wedge G$.

Отметим, что здесь на каждом шаге мы разбиваем задачу на более мелкие задачи; при этом практически для каждой вершины (за исключением тех случаев, когда нам встречаются терминальные вершины) генерируются две новые подзадачи. Таким образом, если две заданные OBDD имеют в сумме n уникальных переменных (если абстрагировать OBDD как некие особого рода бинарные деревья, то результат бинарной операции, примененной к ним, в наихудшем случае имеет n нетерминальных уровней), то в самом плохом случае мы осуществим 2^n разделений, и, как следствие, получим экспоненциальный рост требуемых ресурсов на решение задачи [5]. Отметим, однако, что такова асимптотика осуществления бинарных операций над OBDD лишь при фиксированном порядке переменных в них. Однако сама задача нахождения оптимального порядка для OBDD является NP-полной [4], [5].

Одним из возможных частичных решений описанной выше проблемы сложности вычислений является классический прием, используемый при динамическом программировании – введение так называемого кэша для каждой из бинарных операций. В этом кэше должны храниться некие компактные представления двух OBDD (например, строка, описывающая пути в OBDD к терминальной вершине, помеченной единицей), а также результат операции, примененной к этим OBDD – уже, очевидно, в виде таблицы.

1.5.4. Предикаторные алгоритмы. В этом пункте рассмотрим два алгоритма над OBDD, называемых предикаторными (или алгоритмами осуществления операций квантификации). Первой рассмотрим алгоритм осуществления квантификации существования, или ∃-квантификации.

Пусть $F = F(x, y, z)$ – булева функция трех переменных. Операцией ∃-квантификации F по переменной x назовем такую операцию, результатом которой будет являться функция $G = G(y, z)$, равная единице на тех и только на тех наборах значений (y_1, z_1) переменных (y, z), для которых существует x_1, такое, что на наборе значений (x_1, y_1, z_1) переменных (x, y, z) функция F равна единице (иначе, $F(x_1, y_1, z_1) = 1$) [4]. ∃-квантификацию по нескольким переменным выполняют последовательно по одной переменной в любом порядке [5]. Рассмотренная нами операция ∃-квантификации функции $F = F(x, y, z)$ по переменной x записывается следующим образом: $G(y, z) = (\exists x)\, F(x, y, z)$ [5].

Легко видеть, что $(\exists x)\, F(x, y, z) = F(0, y, z) \vee F(1, y, z)$ [5]. Таким образом, операция ∃-квантификации сводится к операциям подстановки вместо переменной ее значения и дизъюнкции.

Далее рассмотрим операцию осуществления квантификации всеобщности, или ∀-квантификации.

Пусть $F = F(x, y, z)$ – булева функция трех переменных. Операцией ∀-квантификации F по переменной x назовем такую операцию, результатом которой будет являться функция $G = G(y, z)$, равная единице на тех и только на тех наборах значений (y_1, z_1) переменных (y, z), для которых на наборе значений $(0, y_1, z_1)$ и $(1, y_1, z_1)$ переменных (x, y, z) функция F равна единице (иначе, $F(0, y_1, z_1) = F(1, y_1, z_1) = 1$) [4]. ∀-квантификацию по нескольким переменным выполняют последовательно по одной переменной в любом порядке [5]. Рассмотренная нами операция ∀-квантификации функции $F = F(x, y, z)$ по переменной x записывается следующим образом: $G(y, z) = (\forall x)\, F(x, y, z)$ [5].

Легко видеть, что $(\forall x)\, F(x, y, z) = F(0, y, z) \wedge F(1, y, z)$ [5]. Таким образом, и эта операция сводится к рассмотренным ранее: подстановки в функцию вместо переменной ее значения и конъюнкции.

2. ОПИСАНИЕ ПРИЛОЖЕНИЯ

2.1. Постановка задачи. В этом параграфе произведена постановка задачи путем перечисления функциональных и нефункциональных требований, налагаемых на приложение.

2.1.1. Основное требование. Спроектировать и реализовать приложение, позволяющее пользователю проводить формальную верификацию методом проверки моделей без явного задания структуры Крипке. Приложение должно следовать клиент-серверной архитектуре, предоставляя пользователю как можно более широкую возможность использования.

2.1.2. Функциональные требования. Приложение должно предоставлять пользователю следующие возможности:

а) создавать и редактировать проекты – сущности, для которых можно проводить неограниченное число верификаций (в частности, редактирование проектов включает в себя изменение имени, а также активацию/деактивацию);
б) просматривать историю выполнения верификаций для каждого из проектов;
в) получать сведения о результате каждой верификации;
г) вводить компоненты для верификации: программу на формальном языке, набор атомарных предикатов и проверяемое свойство;
д) работать в режиме команды разработчиков: к проектам пользователя должны иметь доступ другие пользователи; кроме того, у них должен быть полный доступ к проектам пользователя, включая создание, редактирование и запуск верификации.

2.1.3. Нефункциональные требования. На приложение накладываются следующие ограничения:

а) используемый формальный язык для ввода пользователем программы должен быть знаком подавляющему большинству программистов;
б) приложение-верификатор должно быть расширяемым как в плане добавления новых возможностей в язык, так и в плане добавления поддерживаемых темпоральных логик;
в) приложение должно работать достаточно быстро; по крайней мере, проблема «комбинаторного взрыва» не должна быть встречающейся в среднем в 90% случаев;
г) приложение не должно тратить больше одного гигабайта оперативной памяти на одну верификацию;

д) приложение как клиент-серверная система должно быть расширяемым в плане добавления новых возможностей в систему: например, создания разграничения прав доступа по ролям пользователей (при этом ролей может быть более трех классических: гость, авторизованный пользователь и администратор/модератор – необходимым может быть учет ролей, специфичных для конкретных проектов);
е) приложение должно быть удобным для сопровождения – что является необходимым, в случае если пользователи в ходе работы с системой найдут в ней ошибки;
ж) приложение (его серверная часть и непосредственно верификатор) должно быть максимально возможно кроссплатформенным.

2.2. Анализ задачи. Приведем в этом пункте анализ задачи, считая приложение единым целым и относя требования, специфичные для определенных модулей приложения, к анализу этих модулей (см. следующие параграфы).

Выбор метода решения этой задачи опирается на материал, изложенный в главе 1. Напомним, что было изложено обоснование выбора метода, основанного на упорядоченных двоичных разрешающих диаграммах (OBDD) (см. п. 1.3). Можно видеть, что этот способ проверки моделей позволяет выполнить некоторые требования, которым должно соответствовать приложение. Кроме того, требование кроссплатформенности и расширяемости сужает круг языков программирования и технологий, которые можно было использовать. Было принято решение остановиться на Java ввиду наличия библиотеки, реализующей хранение OBDD и операции над ними (JavaBDD). Кроме того, использование Java – и, в частности, системы сборки проектов Maven – позволяет реализовать решение как многомодульный проект. Следует отметить, что решение использования многомодульности в организации проекта очевидно: естественным является выделение следующих – как минимум трех – модулей:

а) модуля, содержащего в себе программу-верификатор, которая по пришедшим ей модели на формальном языке, набору атомарных предикатов и проверяемому свойству запускает процесс верификации (проверки свойства на модели), а по его окончании выдает результат;
б) модуля, представляющего собой серверную часть приложения – ту часть, что будет развернута на некоторый сервер приложений;
в) модуля, представляющего собой клиентскую часть приложения.

При использовании системы сборки Maven создание подобной структуры приложения достигается с помощью xml-файла, описывающего процесс сборки [7]. Кроме того, при использовании системы сборки Maven возможным становится установление локальных зависимостей одного модуля на другой [8].

Кроме того, каждый из модулей может быть упакован в любой из возможных поддерживаемых JDK форматов. В частности, при развертывании модуля, содержащего серверную часть приложения, необходимым может оказаться формат WAR выходного файла. Поэтому было принято решение использовать Maven для сборки проекта.

Далее, перейдем к одной из важнейших частей проекта – транспортной части. Очевидным является то, что общение клиента с сервером должно быть унифицированным. В качестве варианта такой транспортной модели можно использовать, например, json-строки. Такой вариант имеет свои плюсы: например, его использование позволяет писать сервер на одном языке программирования, а клиента – на другом. Однако у подобного подхода имеются и минусы. В частности, на наш взгляд, одним из основных неудобств такого подхода является следующее: клиент и сервер неявно устанавливают контракт не только на то, под каким ключами они будут отправлять и принимать определенные значение, но и на тип этих значений. Таким образом, на клиенте и на сервере неизбежно возникает определенная иерархия классов, выполняющих сериализацию и десериализацию некой, также образующей достаточно сложную иерархию, системы классов.

Однако если ограничиться одним языком программирования для клиента и сервера (безусловно, в ряде случаев это не представляется возможным), то задачу общения клиента и сервера можно решить другим путем, который не будет требовать больших усилий на сериализацию и десериализацию объектов. Одним из таких возможных путей – в случае выбора Java технологией написания приложения – является использование для организации клиентской части фреймворка Google Web Toolkit (GWT). Это технология, основанная на трансляции кода, написанного на Java, в язык JavaScript [9]. Решение проблемы организации транспорта выглядит таким: сделать зависимым модуль приложения, служащий серверной частью, от модуля, содержащего клиентскую часть. При этом иерархию классов, используемую для обмена данными между клиентом и сервером, разместить на уровне клиента, написанного на GWT. Таким образом, клиент и сервер будут знать о контракте общения друг с другом лишь из кода, так как будут использовать одни и те же классы (вплоть до местоположения их в пакетах) для отсылки и принятия сообщений. Есть несколько механизмов, позволяющих в рамках использования фреймворка GWT передавать данные с клиента на сервер. Одним из таких механизмов является стандартный механизм, предоставляемый фреймворком GWT – AutoBeanCodex [9]. Однако использование данного механизма требует написания определенного количества громоздкого дополнительного кода, который бы использовался GWT для сериализации и десериализации объектов [9]. Таким образом, плюс GWT-

подхода к передаче данных – минимальное количество утилитарного кода (создания сериализаторов и десериализаторов) – становится едва ощутимым. Поэтому было принято решение остановиться на использовании библиотеки gwt-streamer. Это библиотека требует для сериализации/десериализации объекта (в том числе и с иерархической сложной структурой класса объекта) реализации каждым из используемых классов маркировочного интерфейса Streamable, поддерживая «из коробки» сериализацию/десериализацию базовых классов Java [10]. В своей работе библиотека gwt-streamer использует механизм сериализации/десериализации объектов с помощью Base64 – позиционной системы счисления с основанием 64 [10].

Таким образом, мы приходим к тому, что в качестве технологии, используемой для построения клиента, ориентированного на Web, будет выступать фреймворк GWT. Далее необходимо определиться с тем, как и с помощью чего мы будем строить серверную часть приложения.

Во-первых, следует отметить, что процесс проверки свойства на модели может занимать значительное количество времени. В связи с этим разумно ввести следующее ограничение: в один момент времени пользователь может верифицировать лишь одно свойство на одной модели. При этом сама верификация не должна блокировать основной поток ввода-вывода серверной части, то есть выполняться асинхронно. Однако на клиентскую часть сразу после отправки запроса на верификацию логичным кажется возвращение некоторого ответа: началась ли верификация или не может быть выполнена ввиду того, что данный пользователь уже проверяет некоторое свойство. Иначе говоря, мы приходим к тому, что некоторое свойство объекта, содержащего информацию о пользователе, должно модифицироваться более чем одним потоком.

Во-вторых, требование расширяемости приложения требует от нас такой организации серверной части, при которой изменение или добавление функционала не будет приводить к полному переписыванию системы.

В связи с этими двумя пунктами было решено выбрать следующую классическую модель архитектуры серверной части на Java (рисунок 10):

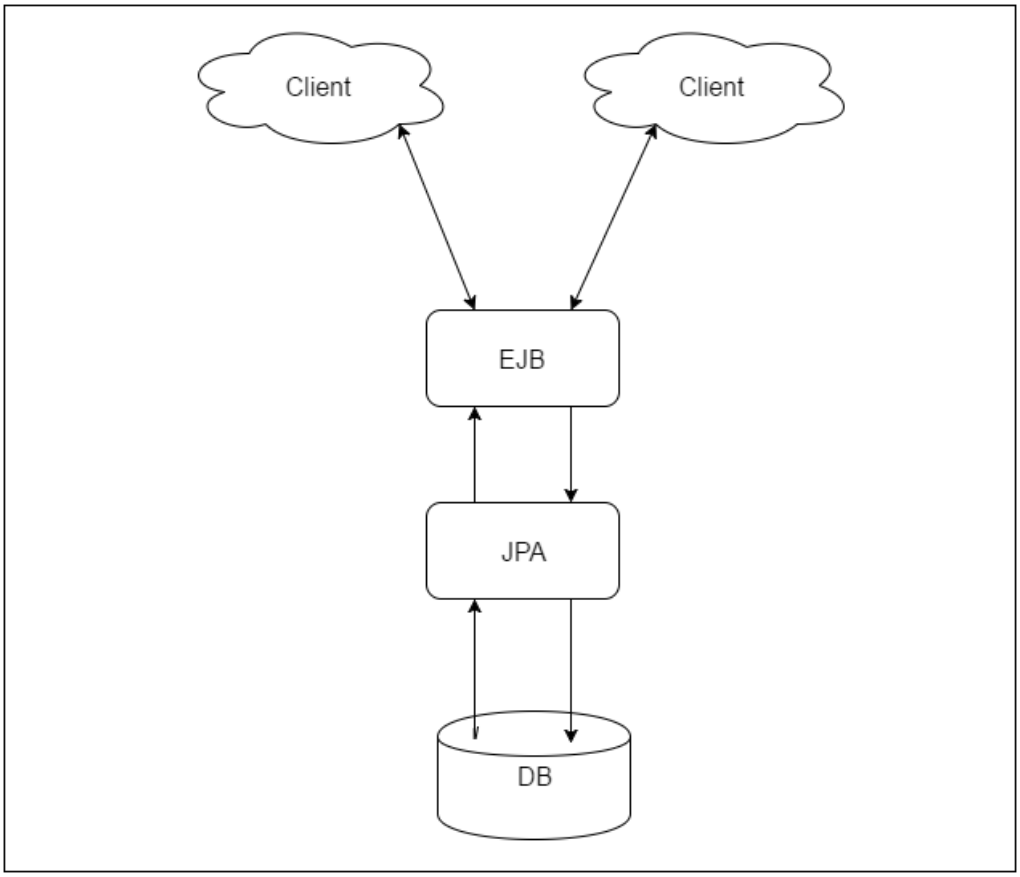

Рисунок 10. Общая архитектура приложения

Здесь именно на уровне EJB можно провести асинхронное выполнение верификации со всеми требованиями, предъявляемыми к нему: стандарт EJB 3.1 включает в себя возможность аннотировать метод или целый класс как Asynchronous [11]. Транзакционный менеджмент и синхронизация потоков осуществляется самой реализацией EJB [11], [12].

В качестве архитектурного решения общения клиентов и сервера логичным кажется использования стиля REST. При использовании описанной выше транспортной модели REST позволит достаточно легко написать приложение-клиент.

Наконец, общение между клиентом и сервером можно реализовать с помощью json-строк, в которых под единственным ключом *message* лежит строка, полученная с помощью применения механизма сериализации библиотеки gwt-streamer к объектам классов транспортной модели.

По результатам анализа задачи мы приходим к выделению трех модулей (модуль-верификатор, серверная часть приложения, клиентская часть приложения), которые должны быть реализованы с помощью языка Java, используя указанные в данном пункте технологии. Кроме того, в рамках знакомства с платформой Android интересным является реализация Android-клиента к серверной части приложения. В нижеследующих параграфах опишем каждый модуль в отдельности.

2.3. Модуль-верификатор приложения. В этом параграфе изложено описание одного из модулей приложения – модуля, выполняющего проверку заданного свойства на заданной модели.

2.3.1. Анализ задачи. Выбор метода решения этой задачи опирается на материал, изложенный в главе 1.

В качестве формального языка для ввода пользователем требующей проверки программы кажется разумным использовать C-подобный язык. Такой язык будет знаком подавляющему большинству пользователей системы. Также необходимо обеспечить подробное логирование работы модуля.

Можно выделить комплекс подзадач, требующих решения в рамках разработки данного компонента приложения:

а) разработка модуля, транслирующего программу на формальном языке в набор OBDD, представляющих структуру Крипке для нее;
б) разработка модуля, транслирующего формулу темпоральной логики в OBDD;
в) разработка самого верификатора, который по имеющимся OBDD структуры Крипке и темпоральной формулы проверяет, удовлетворяет ли программа своей спецификации, а также предоставляет контрпримеры в случае того, если заданное свойство не выполняется на программе.

Проанализируем эти задачи более подробно. Рассмотрим требование под пунктом а). Во-первых, отметим, какие OBDD нам нужны:

а) для представления переходов между состояниями структуры Крипке;
б) для представления состояний структуры Крипке;
в) для представления стартовых состояний структуры Крипке;
г) набор в количестве имеющихся атомарных предикатов для представления состояний структуры Крипке, в которых верен каждый атомарный предикат.

Во-вторых, определим конструкции языка. Для первой итерации разработки приложения достаточно небольшого языка. Пусть он будет идентичен тому, что описывается в п. 1.4.5.

В-третьих, рассмотрим процесс трансляции программы на определенном нами языке в структуру Крипке с представлением вышеперечисленными OBDD. Итак, сначала необходимо *разметить* программу – аналогично тому, как мы это делали в п. 1.2. Затем по правилам, изложенным в п. 1.4.5., *транслировать* программу в OBDD, выполняя полный перебор всех значений переменных в случае ее считывания.

В-четвертых, заметим, что процесс трансляции программы в OBDD требует модуля, способного вычислять выражения и проверять истинность условий. Для этого требуется модуль, способный переводить выражение и условие в обратную польскую нотацию. Отметим, что выражение можно считать в некотором роде частным случаем условия – ведь в условии могут сравниваться, например, результаты двух выражений. Поэтому логично создать один модуль, *переводящий условие в обратную польскую запись*, а также один *вычислитель* на стеке. Кроме того, атомарные предикаты по своей структуре аналогичны условиям, использующимся в предложении *if*. Поэтому здесь мы получаем двойную выгоду: создав модуль для проверки истинности условий, мы уже будем иметь *модуль для проверки истинности атомарных предикатов*.

Далее перейдем к анализу второй подзадачи (пункт б). Формулу темпоральной логики необходимо сначала перевести в обратную польскую нотацию. Для этого нужен отдельный модуль. При этом на данном шаге должна происходить минимальная *валидация* введенной пользователем формулы. Поэтому для каждой логики должен быть свой *модуль переведения формулы в обратную польскую запись*. Далее, формула, выражающая спецификацию, в качестве аргументов принимает атомарные предикаты. Встает вопрос о том, как пользователь должен указывать эти аргументы. Логичным кажется следующее решение: пусть атомарные предикаты имеют порядковый номер, начиная с 0 (соответствующую информацию на уровне клиента необходимо предоставить пользователю при вводе им атомарных предикатов). Тогда применение темпорального или логического оператора к атомарному предикату будет вводиться следующим образом, например: CTL-формула $EF(0\ AND\ 1)$ обозначает, что «из стартового состояния существует некоторый путь, на котором в некотором состоянии верны атомарные предикаты с порядковыми номерами 0 и 1».

Теперь перейдем к вопросу вычисления формулы темпоральной логики. Отметим, что модуль, который выполняет это, должен иметь доступ к набору OBDD, представляющих атомарные предикаты. При этом заметим, что до начала вычисления формулы темпоральной логики OBDD атомарных предикатов уже должны быть получены, равно как и OBDD, представляющие структуру Крипке (см. п. 1.4.4).

Наконец, обратимся к третьей подзадаче. Очевидно, что модуль, который выполняет верификацию, должен вступать в работу после того, как получены OBDD для формулы темпоральной логики и структуры Крипке. При этом функция, выполняющая верификацию, должна возвращать одно из трех значений: свойство выполняется, свойство не выполняется, произошла ошибка. Если свойство не выполняется, этот же модуль должен давать возможность получе-

ния контрпримеров. При этом, так как контрпримеров может быть очень много, следует ограничить их максимальное число.

Кроме того, необходим модуль, который все эти модули свяжет воедино и будет «общаться» с внешней системой (например, с серверной частью приложения) – *контроллер*.

Надо отметить, что, ввиду вероятной громоздкости вычислений, все они должны быть *ленивыми*, то есть запускаться не заранее, а только тогда, когда требуется их результат.

2.3.2. Реализация. В этом параграфе подробно рассмотрим реализацию приложения, приводя UML-диаграммы и описывая возникшие проблемы и пути их решения.

Модуль разбит на 4 пакета: *com.system*, *com.system.util*, *com.system.temporallogic*, *com.system.kripkestructure*. В последних двух пакетах находятся классы, обеспечивающие трансляцию соответственно формулы темпоральной логики и программы на формальном языке в OBDD. Первый пакет хранит в себе модули, отвечающие за верификацию, а также и некоторые компоненты, общие для всего приложения. Во втором расположены утилитарные классы: класс, обеспечивающий корректную работу логирования. Сначала представим классовую диаграмму пакета *com.system.util* (рисунок 11):

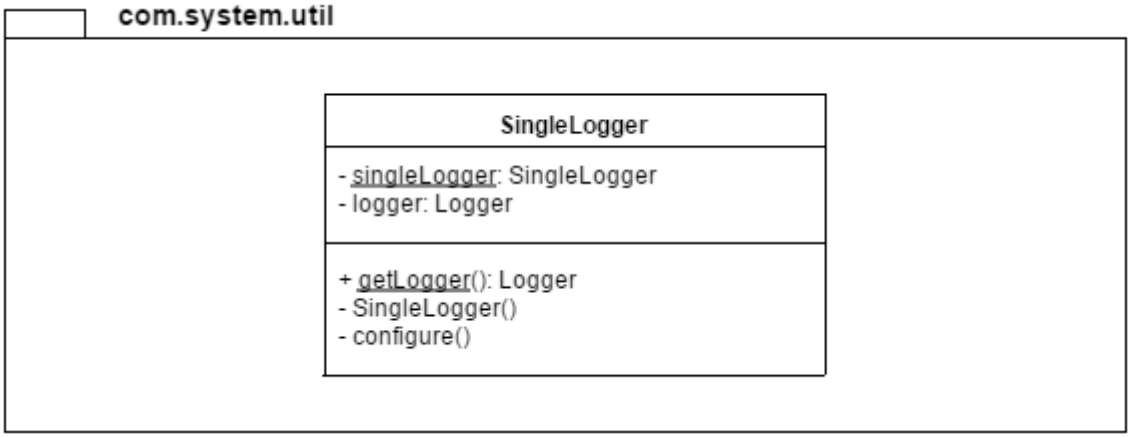

Рисунок 11. Диаграмма классов com.system.util

Здесь, как следует из названия, SingleLogger представляет собой обертку над классом Logger, реализуя паттерн Singleton.

Далее, представим диаграмму классов пакета *com.system* (рисунок 12):

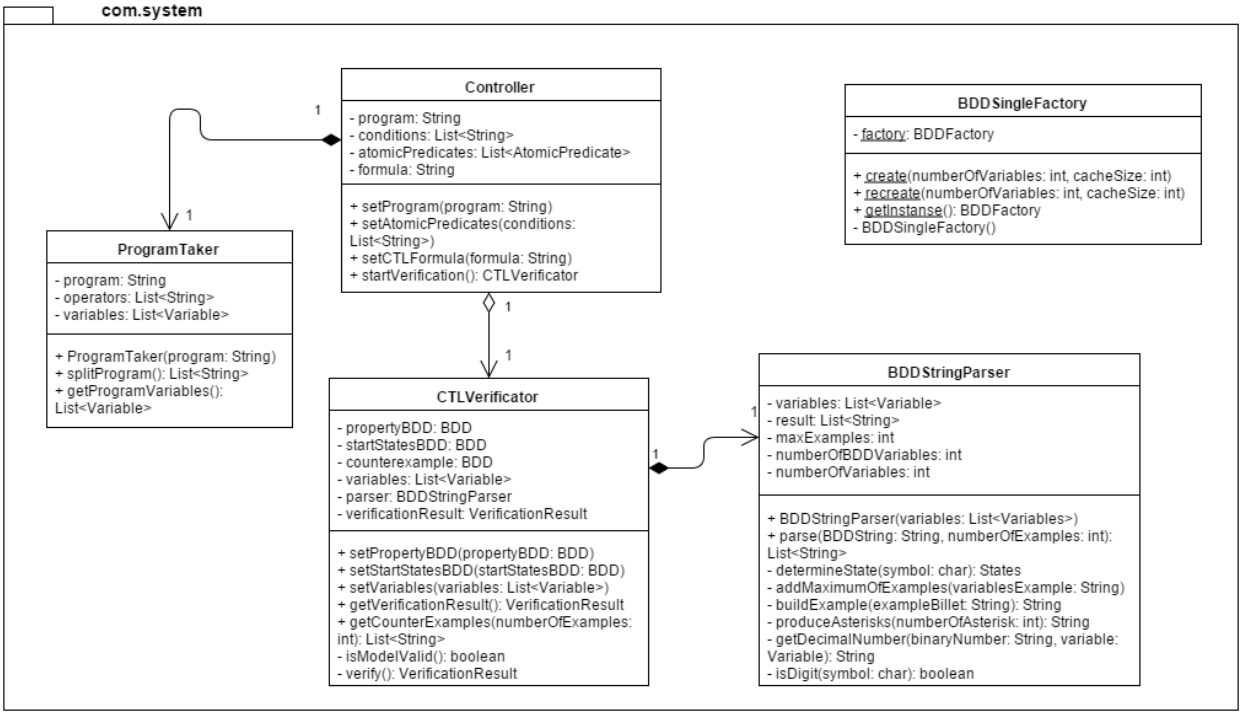

Рисунок 12. Диаграмма классов com.system

Класс BDDSingleFactory представляет собой обертку над абстрактным классом BDDFactory использованной библиотеки JavaBDD. Название класса Controller говорит само за себя: этот класс фактически «собирает» все приложение в единое целое, связывая между собой функционал пакетов, получает данные от внешней системы и предоставляет ей результат верификации. Класс ProgramTaker преобразует программу в список строк, а также определяет все программные переменные. Поясним подробнее, какого вида последовательности символов выделяются в элемент списка:

а) строка присваивания (*<имя_переменной> = <выражение>*);
б) строка считывания (*read(<имя_переменной>)*);
в) арифметическое выражение;
г) строка if (*if (<условие>)*);
д) строка else (*else*);
е) строка while (*while (<условие>)*);
ж) всякая открывающая фигурная скобка (*{*);
з) всякая закрывающая фигурная скобка (*}*).

Кроме того, возможно попадание в список пустых строк. Заметим, что все строки в списке хранятся без начальных и конечных пробелов.

Класс CTLVerificator по имеющимся OBDD, представляющим стартовые состояния структуры Крипке и заданную пользователем CTL-формулу, проверяет, соответствует программа введенной спецификации или нет. Его метод *verify* работает по алгоритму, изложенному в п. 1.4.6. В случае, если свойство

не выполняется, необходимо представить контрпримеры. Алгоритм их нахождения также изложен в п. 1.4.6. Класс BDDStringParser конструирует такие контрпримеры, обрабатывая строковое представление объекта класса BDD.

Теперь перейдем к рассмотрению пакета *com.system.temporallogic* (рисунок 13):

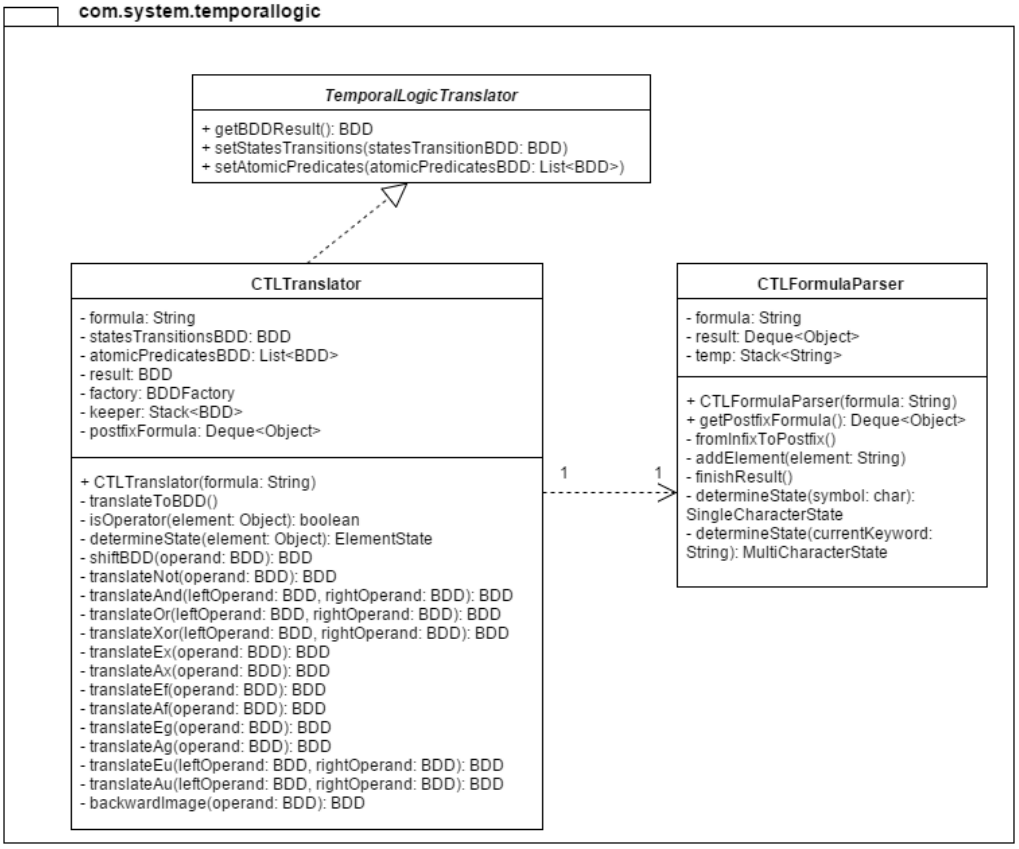

Рисунок 13 Диаграмма классов com.system.temporallogic

Здесь CTLTranslator реализует интерфейс TemporalLogicTranslator. Объекты, имеющие тип такого интерфейса, являются трансляторами формулы темпоральной логики в OBDD. На данный момент поддерживается только CTL; однако, принятое решение позволит без труда обеспечить поддержку, скажем, LTL. Класс CTLTranslator работает по алгоритму, изложенному в п.1.4.4. Класс CTLFormulaParser переводит формулу CTL из инфиксного в постфиксный вид, выполняя некоторую валидацию формулы. При разработке приложения было принято решение, что темпоральный оператор U принимает квантор общности или существования (A или E) непосредственно перед символом U, а не перед всем выражением, как принято писать (принято писать, например, $A(\varphi\ U\ \mu)$, а мы принимаем лишь формулу вида ($\varphi\ AU\ \mu$)).

Наконец, рассмотрим пакет *com.system.kripkestructure* (рисунок 14):

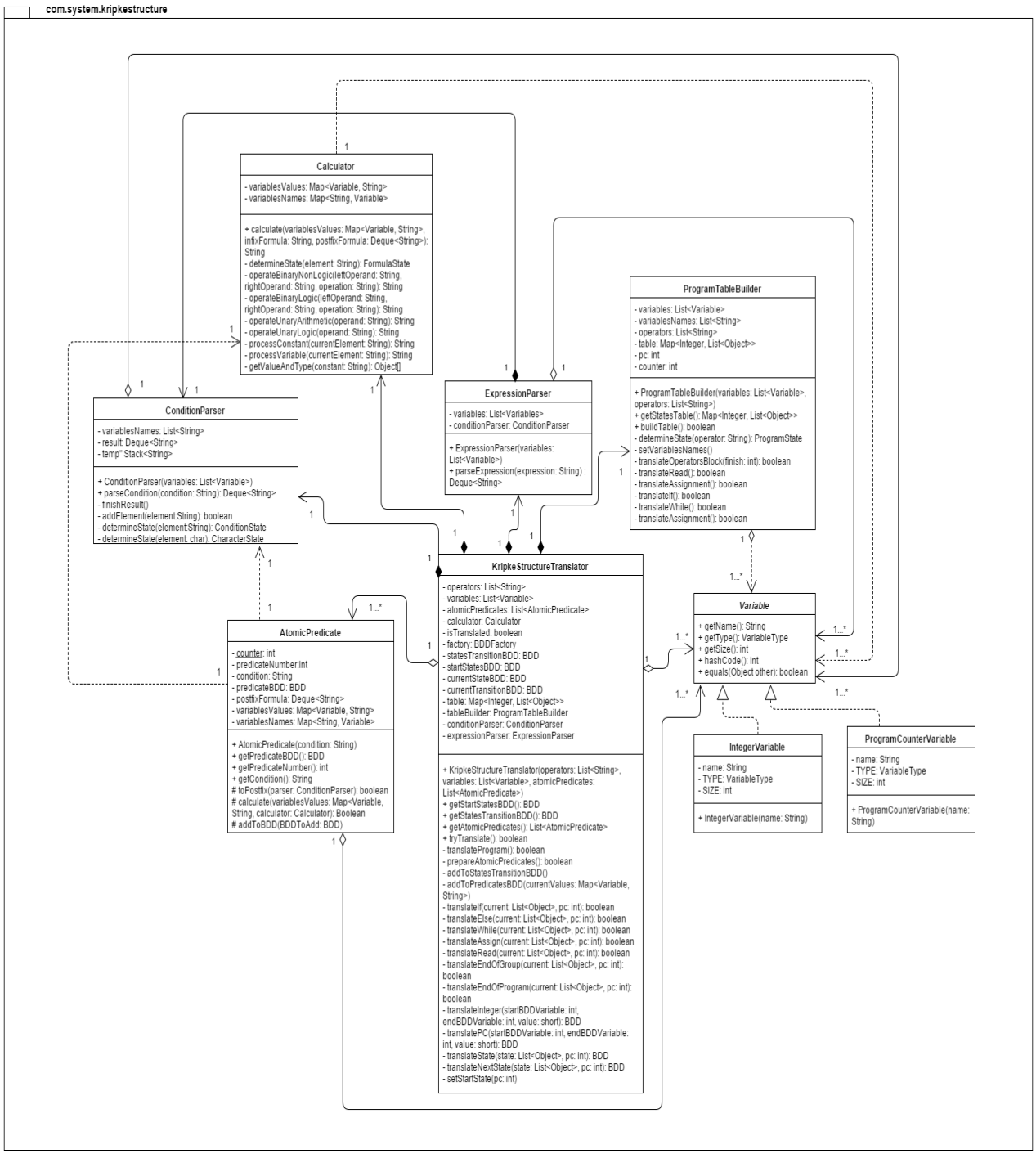

Рисунок 14. Диаграмма классов com.system.kripkestructure

Здесь ключевым является класс KripkeStructureTranslator, строящий набор OBDD для представления структуры Крипке введенной пользователем программы (экземпляры класса BDD библиотеки JavaBDD, представляющие множество стартовых состояний, отношение переходов и состояния, в которых верен каждый из атомарных предикатов). Перед началом работы объект этого класса создает объект класса ProgramTableBuilder. Он, в свою очередь, переводит список строк программы – о котором упоминалось выше – в специального вида таблицу. Объясним, как выглядит эта таблица.

Пусть имеется N программных переменных. Первые N ячеек строки таблицы содержат в себе выражения, характеризующие текущее состояние программы. Рассмотрим, какими могут быть эти выражения:

а) UNDEFINED. Это выражение означает, что переменная лишь объявлена, но никакого значения ей еще не присваивалось.
б) DEFINED. Это выражение означает, что переменная была прочитана – то есть, ее значение вполне определено, но нам не известно.
в) FIXED. Это выражение означает, что в данном состоянии закончено ветвление, и в зависимости от того, выполнено ли условие в *if* или *while,* переменная примет то или иное значение.
г) обычное выражение языка, подобное тому, что стоит по правую сторону оператора присваивания. Этот тип выражения является предикатом и показывает, каково значение переменной.

Следующие N ячеек таким же образом характеризуют состояние, в которое мы переходим. $(2N + 1)$-ая ячейка содержит порядковый номер текущего состояния – значение переменной *ProgramCounter* (см. п. 1.4.5). $(2N + 2)$-ая ячейка содержит значение *ProgramCounter*, отвечающее следующему состоянию – тому, в которое мы переходим. Наконец, последняя ячейка содержит комментарий:

а) в случае если текущая строка есть считывание переменной – строку "*READ J*", где *J* – номер считываемой переменной (переменные нумеруются с 0);
б) в случае если текущая строка есть операция присваивания – строку "*ASSIGN J*", где *J* – номер присваиваемой переменной;
в) в случае если текущая строка имеет вид *if (<условие>)* – строку "*IF <условие>;ELSE <номер строки, к исполнению которой нужно перейти, если <условие> неверно>*";
г) в случае если текущая строка есть *else* – строку "*ELSE <номер строки, в которой содержится if данного else>*";
д) в случае если текущая строка имеет вид *while (<условие>)* – строку "*WHILE <условие>; ELSE <номер строки, к исполнению которой нужно перейти, если <условие> неверно>*";
е) в случае если текущая строка есть закрывающая фигурная скобка – строку вида "*<IF | ELSE | WHILE> <номер строки, предшествующей открывающей фигурной скобке для данной закрывающей> ENDS*".

Отметим, что для упрощения обработки программы было принято решение о том, что всякий набор выражений, относящихся к *if, else* и *while*, должен быть заключен в фигурные скобки.

После формирования таблицы начинает свою работу класс KripkeStructureTranslator. Перед началом трансляции программы в OBDD происходит «приготовление» атомарных предикатов: условия предикатов переводятся в обратную польскую нотацию (с помощью класса ConditionParser). Далее определяется, какого типа первая строка таблицы. В зависимости от того, какое выражение представляет эта строка, управление передается определенному методу: *translateIf*, *translateElse*, *translateWhile*, *transalteAssign*, *translateRead*, *translateEndOfGroup*, *translateEndOfProgram*. Каждый из этих методов (за исключением последнего) определяет, каков тип следующей строки и передает управление соответствующему методу. Вычисление условий происходит с помощью классов ConditionParser и Calculator, выражений –ExpressionParser и Calculator.

Фактически, внутри KripkeStructureTranslator происходит полный перебор всех возможных состояний программы одновременно с ее пошаговым выполнением: предикаты из строк таблицы переводятся в конкретные состояния.

Далее, покажем, как классы из разных пакетов связаны между собой (рисунок 15):

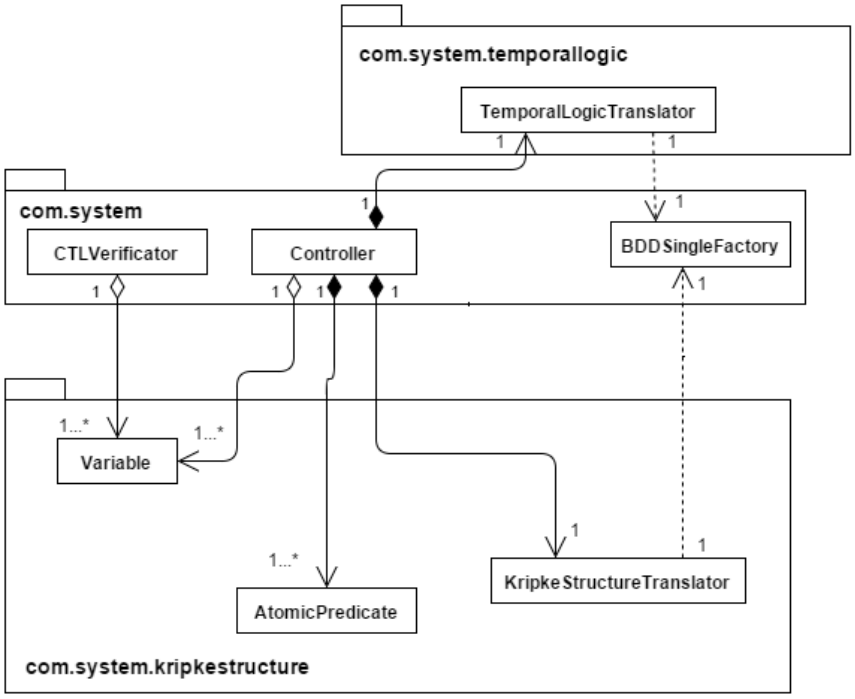

Рисунок 15. Общая диаграмма классов модуля-верификатора

Здесь опущены связи внутри пакетов и показаны лишь публичные классы (связи внутри пакетов можно увидеть на диаграммах выше).

Далее, сделаем некоторые замечания.

Во-первых, на данный момент в программе представлены лишь два типа переменных: IntegerVariable и ProgramCounterVariable.

Во-вторых, выбор библиотеки для работы с OBDD на Java внес некоторые корректировки в реализацию. Изначально планировалось,

что IntegerVariable будет представлять собой 32-битный тип данных, как и ProgramCounterVariable. Однако после того, как потребление оперативной памяти для на тесте, где в программе была одна переменная, превысило 2 гигабайта и периодически возникало исключение *Java Heap Space*, было принято решение сократить размер IntegerVariable до 16 бит, как и ProgramCounterVariable. Однако тот же тест по-прежнему вызывал те же самые ошибки. Сокращение размера ProgramCounterVariable до 8 бит решило проблему, уменьшив размер потребляемой оперативной памяти до 1 гигабайта. Эта проблема была обнаружена на этапе тестирования. Таким образом, программа, задаваемая пользователем, не может быть длиннее 128 строк.

В-третьих, удалось уменьшить количество потребляемой оперативной памяти за счет использования нативного кода – библиотеки BuDDy, написанной на C. Библиотека JavaBDD по сути представляет собой аналог этой библиотеки, но так же позволяет использовать BuDDy для работы. Однако данный модуль при таком решении может запуститься лишь на JVM x86, так как BuDDy собрана именно под такую архитектуру.

В-четвертых, запуск сложных тестов для этого модуля на операционной системе семейства Windows (использовалась Windows 10) оказался неудачным. Размер запрашиваемой памяти для сложных тестов составляет 1 гигабайт. JVM для Windows реализована так, что требует нефрагментированную часть RAM для своей работы. Тесты проводились на ноутбуке с 6 гигабайтами оперативной памяти, однако ни один из них не был удачен. Поэтому было принято решение запустить приложение на UNIX-подобных операционных системах. В качестве такой системы была выбрана Ubuntu 14.04 LTS. Запуск тестов прошел успешно, проблема была решена.

Однако описанное решение нарушает требование кроссплатформенности модуля-верификатора (и, как следствие, всей серверной части приложения). Так как очевидным узким местом компонента-верификатора является библиотека для работы с OBDD, интересной является задача анализа решения, написанного в библиотеке JavaBDD – а также анализ тех функций этой библиотеки, которые необходимы модулю-верификатору для работы. В следующем пункте опишем, какие выводы относительно этой библиотеки были сделаны и какие результаты получены.

2.3.3. Реализация библиотеки работы с OBDD. После проведенного анализа исходного кода как библиотеки BuDDy, так и библиотеки JavaBDD был сделан вывод о не самой оптимальной структуре хранения, использующейся в этих библиотеках. Архитектура этой библиотеки представлена ниже (рисунок 16):

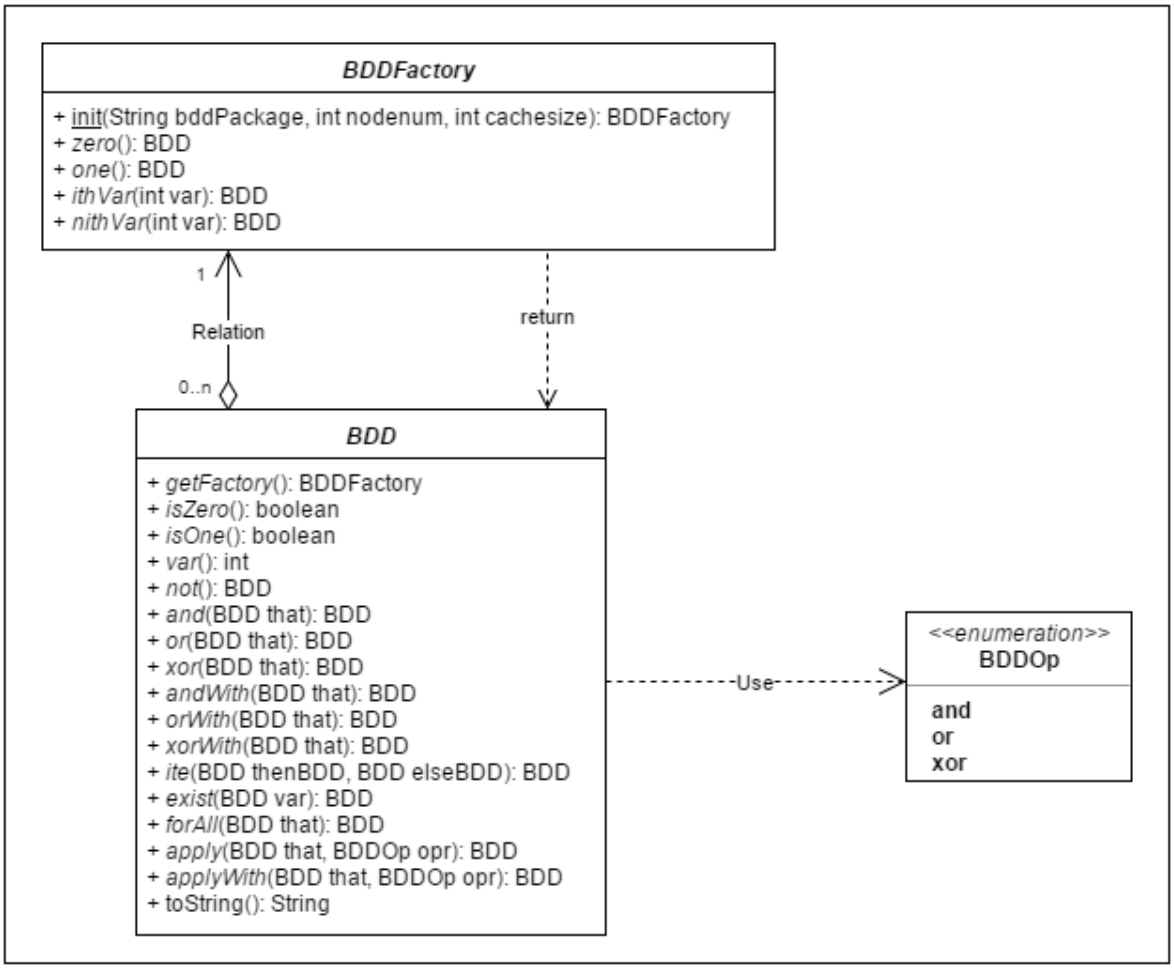

Рисунок 16. Диаграмма классов библиотеки для работы с OBDD

Оговоримся сразу, что на диаграмме приведены далеко не все классы и их методы, представленные в библиотеке, а лишь только те, что используются модулем-верификатором. Здесь класс BDDFactory, что следует из его названия, является фабрикой для объектов типа BDD. При этом статический метод *init*, принимая на вход полное имя класса-наследника BDDFactory, а также дополнительные параметры – максимальное количество переменных и размер кэша – создает объект соответствующего типа. С каждым конкретным наследником BDDFactory связывается реализация абстрактного класса BDD, объекты которой и создает фабрика. При этом проведение логических и предикаторных операций разрешено только над объектами BDD, порожденным одним и тем же объектом BDDFactory.

Как реализация этой абстрактной пары через язык Java, так и через C использует одну и ту же структуру хранения OBDD. OBDD хранятся в таблице, подобной той, что описана в пункте 1.5.1. Однако таблица эта хранится не на уровне класса-наследника BDD, а на уровне класса-наследника BDDFactory, причем таблица эта является также глобальным кэшем для всех BDD, порожденных одним объектом класса BDDFactory. Это обеспечивает сравнительно быструю работу, однако затрудняет очистку такой таблицы, что приводит к проблемам с количеством расходуемой памяти, описанным выше.

В итоге было принято решение выделить еще один модуль клиент-серверного приложения – собственную имплементацию абстрактной пары BDDFactory–BDD, вынеся хранение таблицы, отвечающей за представление OBDD в памяти компьютера, на уровень класса-наследника BDD.

Диаграмма классов написанной реализации представлена ниже (рисунок 17):

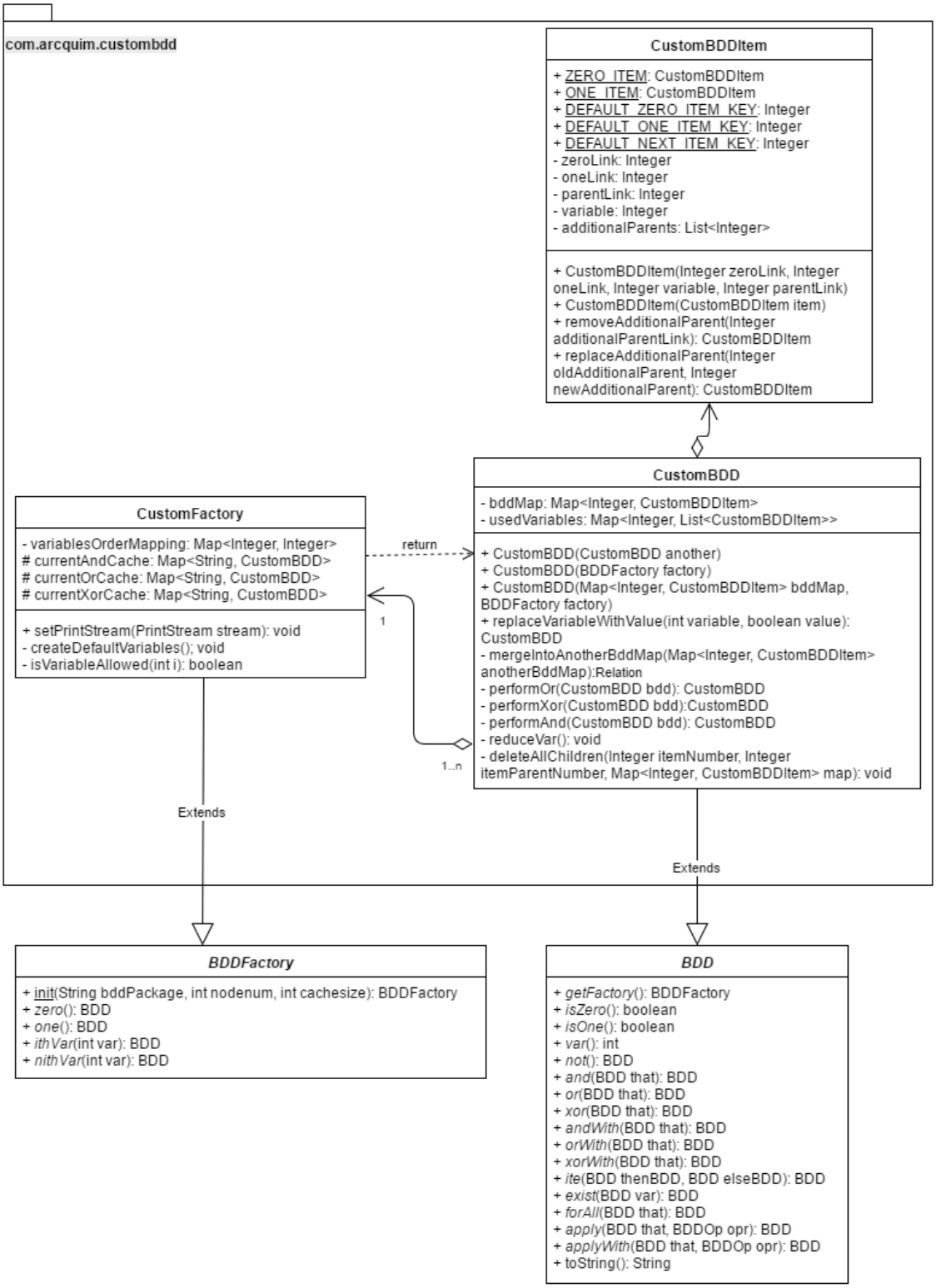

Рисунок 17. Диаграмма классов реализации библиотеки работы с OBDD

CustomFactory и CustomBDD реализуют далеко не все абстрактные методы классов BDDFactory и BDD; для данной задачи было решено ограничиться реализацией лишь тех методов, которые требуются для работы модуля-верификатора; так, например, не реализуются методы, позволяющие проводить переупорядочивание переменных в OBDD. Рассмотрим процесс работы этой частичной реализации библиотеки.

При инициализации объекта класса CustomFactory создается два кэша объектов CustomBDD для булевских переменных под номерами от 0 до nodenum; копии объектов из кэша можно получить, вызвав методы *ithVar* и *nithVar* (если отождествить булевскую переменную под номером 0 с, например, булевской переменной *x*, то ithVar(0) вернет объект CustomBDD, представляющий функцию $f(x) = x$, а nithVar(0) – объект CustomBDD, соответствующий функции $f(x) = \bar{x}$). При этом при передаче параметра типа *int*, соответствующего номеру булевской переменной, в любую из этих двух функций, сначала происходит его валидация (метод *isVariableAllowed*). В случае если ее результат есть *false*, методы *ithVar* и *nithVar* выбрасывают *IllegalArgumentException*. Объекты CustomBDD, представляющие тождественно ложную или истинную функцию, можно получить, вызвав методы *zero* и *one* объекта CustomFactory соответственно. Далее, в реализацию CustomFactory заложена возможность расширения класса так, чтобы он поддерживал переупорядочивание булевских переменных (поле variablesOrderMapping). Также на уровне CustomFactory находятся кэши трех поддерживаемых бинарных операций:

а) *currentAndCache* – операции and;
б) *currentOrCache* – операции or;
в) *currentXorCache* – операции xor.

Кэширование реализовано так, что кэш отдельно взятой операции очищается, как только количество элементов в нем превысит 100. При этом в кэше хранится соответствие двух OBDD результату бинарной операции над ними (хранится соответствие результата конкатенации строковых представлений двух объектов класса CustomBDD результату соответствующей бинарной операции над OBDD, соответствующими этим объектам).

Класс CustomBDDItem является классом, описывающим структуру элементарной единицы CustomBDD (фактически вершины графа). Поле, являющееся объектом класса List, параметризованного классом Integer, описывает родителей терминальных вершин – вершин, представляющих ноль и единицу (см. п. 1.4.1, п. 1.5.1).

Теперь перейдем к рассмотрению класса CustomBDD. Он предоставляет три конструктора: конструктор копирования, конструктор, принимающий лишь объект CustomFactory, и конструктор, принимающий объект класса Map, пара-

метризированный классами Integer и CustomBDDItem. Фактически, поле *bddMap* является таблицей, о которой идет речь в п. 1.5.1. При создании объекта класса CustomBDD вместе с *bddMap* создается и заполняется поле *usedVariables*, где под индексом булевской переменной лежит список объектов CustomBDDItem, соответствующих вершинам для каждой из переменных. Метод *var*, определенный в классе BDD библиотеки JavaBDD, возвращает значение, идентифицирующее, какая переменная соответствует головной вершине графа. Метод *replaceVariableWithValue* работает по алгоритму, описанному в п. 1.5.2; для удаления ненужных частей графа используется метод *deleteAllChildren*. Метод *not* – построение отрицания функции – работает по соответствующему алгоритму из п. 1.5.3. Наконец, бинарные операции – методы *apply applyWith* – работают следующим образом: методы *or* (*orWith*), *and* (*andWith*), *xor* (*xorWith*) обращаются к методу *apply* (*applyWith*) с соответствующими аргуметами. Методы *apply* и *applyWith*, в зависимости от пришедших к ним аргументов, вызывают приватные методы *performOr*, *performAnd* или *performXor*. Эти же методы реализуют алгоритм осуществления бинарной операции, описанный в п. 1.5.3. При вызове каждого из этих методов проверяется, нельзя ли уже определить результат операции (или операция тривиальна – например, мы пытаемся вычислить значение логического «или» над тождественным нулем и некой функцией – или результат применения текущей операции содержится в кэше). Если результат неясен, определяется, какая из двух OBDD – соответствующая текущему объекту или аргументу – имеет в качестве головного элемента вершину, соответствующую переменной с меньшим порядком (вызывается метод *var* у обоих объектов). В зависимости от этого вместо головной вершины текущего объекта и/или аргумента функции подставляются значения *true* и *false* (вызывается метод *replaceVariableWithValue*). Затем происходит два рекурсивных вызова текущего метода (*performOr*, *performAnd* или *performXor*) – у объекта CustomBDD, имеющего меньшее значение *var*, полученного подстановкой вместо головной переменной *false*, с объектом CustomBDD, полученным из второго объекта путем подстановки вместо головной переменной аргумента значения *false*, в качестве аргумента – и аналогичный вызов, где в каждую из двух OBDD подставлено значение *true*. После этого вызывается функция *ite* (которая, в свою очередь, вызывает приватный метод *ifThenElse*) у объекта с меньшим значением *var*, где в качестве первого аргумента выступает результат первого рекурсивного вызова, а в качестве второго аргумента – второго рекурсивного вызова. После этого результат этой функции заносится в кэш соответствующей бинарной операции, а затем возвращается в качестве результата. Метод же *ifThenElse* выполняет слияние, описанное в п. 1.5.3.

Наконец, методы *exist* и *forAll* класса CustomBDD осуществляют описанные в п. 1.5.4 предикаторные операции – квантификации существования и всеобщности соответственно. При этом необходимо отметить, что аргумент здесь трактуется как перечисление переменных, по которым следует произвести предикаторную операцию.

В итоге написанная частичная реализация библиотеки работы с OBDD была подставлена в класс BDDSingleFactory пакета com.system. Тесты были успешно завершены, требуя не более 512 Мб оперативной памяти; однако время их выполнения выросло вдвое, что объясняется уменьшением роли кэширования при реализации. Кроме того, фактически мы избавились от библиотеки JavaBDD, а кроме того, сняли ограничения на версию архитектуры JDK и используемую операционную систему.

2.4. Модуль серверной части приложения. В этом параграфе рассмотрим модуль, представляющий из себя серверную часть приложения – модуль, архитектура которого фактически представлена на рисунке 10 в п. 2.2.

2.4.1. Анализ задачи. Как было упомянуто в общем анализе задачи (п. 2.2), следует остановиться на трехуровневой организации серверной части приложения: база данных, ORM (JPA), слой бизнес-логики (EJB). Исходя из постановки задачи, определим сущности для нашего приложения.

Очевидно, что требование организации многопользовательской системы влечет за собой создание сущности «пользователь». Чтобы избежать совпадения названия таблицы и ее колонок с ключевыми словами, используемыми в большинстве реляционных СУБД, назовем эту сущность *systemclients*. Далее, требование наличие проектов у пользователя, очевидно, влечет за собой наличие сущности «проект». Дадим этой сущности название *project*. Из требования просмотра истории верификации для каждого проекта вытекает наличие сущности «история проекта». Будем называть ее *projecthistory*. Наконец, нам потребуется словарь, который будет хранить возможные результаты верификации; для упрощения построения объектно-реляционного соответствия сделаем этот словарь отдельной сущностью с названием *result*.

Далее, так как существует требование работы над проектом команды разработчиков, то необходимо организовать некий общий доступ к проектам. К сожалению, грамотное построение политики доступа к защищенным ресурсам – тем более в таком случае, когда один пользователь для одних сущностей (проектов) может быть администратором, для вторых – пользователем, а к третьим вообще не иметь доступа – требует значительного количества времени. Основной целью же разрабатываемого приложения служит предоставление пользователю возможности проверить свойство на модели. Таким образом, разделение доступа к ресурсам можно считать второстепенной задачей.

Наконец, пользователь, согласно требованиям, должен получать уведомления о том, как завершилась запущенная над его проектом верификация. При этом, как было отмечено в п. 2.2, разумно позволить пользователю в один момент времени осуществлять лишь одну верификацию – в противном случае, так как символьная верификация может занимать достаточно большое количество памяти и времени, у операционной системы, обслуживающей серверную часть приложения, может не хватить ресурсов. Однако, сразу после того, как пользователь отправляет модель и свойство на проверку, разумно сообщить пользователю, началась ли проверка или нет. В качестве решения логичным кажется использование асинхронного метода верификации, о чем говорилось в п. 2.2. Тем не менее, одним из важных является вопрос: как сообщить пользователю о результате верификации? Одним из самых очевидных решений является отсылка пользователю на почту результата верификации вместе с моделью и свойством. Таким образом, мы приходим к необходимости использования JavaMail, используя для отправления писем, например, аккаунт на популярном почтовом сервисе Gmail.

Таким образом, мы приходим к тому, что для каждого пользователя системы должен храниться адрес его электронной почты. Так как задача разграничения доступа к ресурсам не является первоочередной, можно остановиться на следующем решении. Для входа в систему пользователю достаточно ввести валидный e-mail, после чего он получает доступ к проектам пользователя с данным адресом электронной почты – если они есть; если же проектов у пользователя с введенным адресом почты нет, то на экране клиентского приложения должно отобразиться соответствующее сообщение.

Далее, определим, какие возможные значение имеет словарь результатов верификации. Не считая модуль-верификатор безупречно и безошибочно реализованным, остановимся на следующих четырех результатах:

а) свойство выполняется;
б) свойство не выполняется;
в) невалидные входные данные;
г) ошибка в ходе верификации.

Теперь решим, какую из реляционных СУБД следует использовать. Следует отметить, что требование кроссплатформенности приложения делает невозможным использование MS SQL Server. Использование СУБД Oracle является излишним при построении первой, фактически прототипной версии серверной части. Наиболее разумным кажется выбор PostgreSQL в качестве СУБД – мощной, быстрой, кроссплатформенной и достаточно легкой для настройки.

Наконец, необходимо определить, какой сервер приложений использовать. Требование расширяемости приложения – а значит, потенциально больше

число используемых возможностей Java EE – а также академический интерес делают привлекательным использование сервера приложений WildFly 10.

2.4.2. Реализация. Перейдем к описанию реализации серверной части приложения. Опираясь на выводы в предыдущем пункте, приведем ER-диаграмму базы данных приложения (рисунок 18):

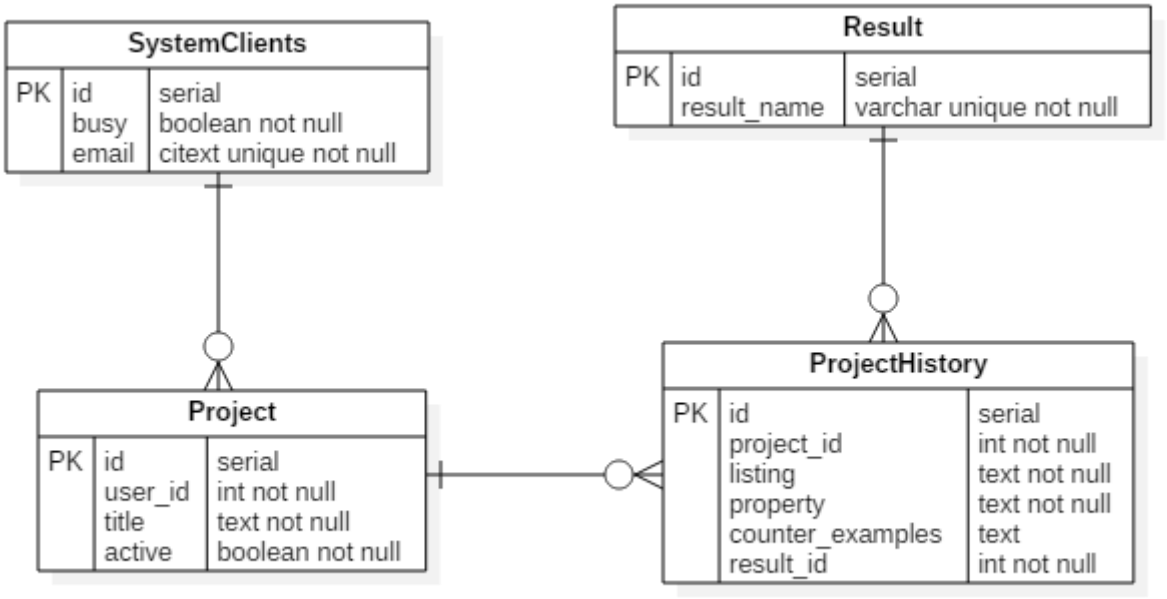

Рисунок 18. ER-диаграмма предметной области

Здесь атрибут *busy* сущности SystemClients говорит о том, можно ли в данный момент выполнять верификацию над проектами пользователя. Атрибут *user_id* сущности Project, являясь обязательным внешним ключом, говорит о том, на чей e-mail будут отправляться нотификации о проверке моделей. Атрибуты *project_id*, *result_id* сущности ProjectHistory, являясь обязательными внешними ключами на сущности Project и Result соответственно, обеспечивают понимание того, для какого проекта и с каким результатом выполнялась верификация.

Скрипт создания базы приведен в приложении 3. Следует отметить, что для валидации адреса электронной почты применяется функция *Email::Address* расширения PL/Perl PostgreSQL [13]. Тип *citext* поля *email* таблицы SystemClients также доступен как расширение PostgreSQL, являясь не чувствительным к регистру символов аналогом типа данных *text* [14].

В качестве имплементации стандарта JPA было принято решение использовать EclipseLink. На следующей диаграмме классов покажем, как реализована серверная часть, опуская классы-сущности, а также транспортную часть модуля, которая будет рассмотрена в п. 2.5 (рисунок 19):

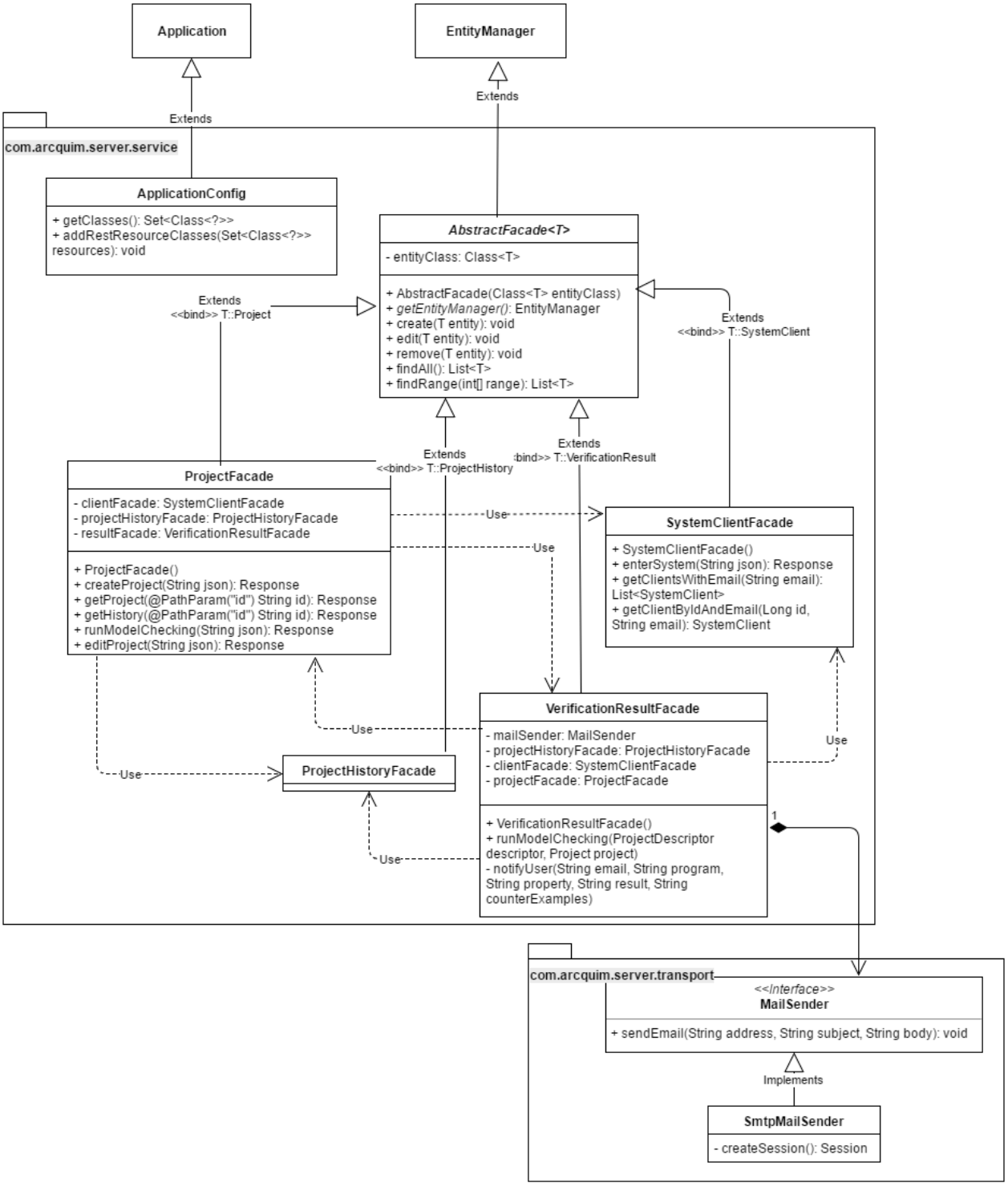

Рисунок 19. Диаграмма классов серверной части приложения

Здесь в пакете *com.arcquim.server.service* находятся классы, представляющие собой слой бизнес-логики приложения. В пакете *com.arcquim.server.transport* лежат классы, специфичные для транспортной части: отправка e-mail и те классы, что отвечают за сериализацию и десериализацию объектов в и из json-строки, описанной в п. 2.2 (подробнее о транспортной модели рассказано в п. 2.5).

На уровне пакета *com.arcquim.server.service* выполняется асинхронная проверка модели: метод *runModelChecking* ProjectFacade выполняет вызов од-

ноименного метода VerificationResultFacade, объект которого попадает в первый с помощью механизма Dependency Injection EJB. При этом метод *runModelChecking* класса VerificationResultFacade помечен аннотацией *Asynchronous*, что делает возможным такой вызов встроенным механизмом фреймворка EJB. Перед делегированием выполнения верификации объекту класса VerificationResultFacade объект ProjectFacade проверяет, выполняется ли сейчас верификация для пользователя (значение поля *busy* сущности SystemClient). Если пользователю разрешено проводить верификацию, поле busy объекта SystemClient выставляется в *true*, происходит коммит, после чего объект класса VerificationResultFacade вызывает соответствующие методы класса Controller пакета *com.system*. После завершения верификации происходит отправка пользователю электронной почты и выставление поля *busy* объекта класса SystemClient в *false*.

Слой EJB-бинов, используя API JAX-RS, определяет REST API сервера. Отметим, что сам серверный модуль развертывается на WildFly с контекстом *system*, REST API же имеет префикс *webresources*. Приведем далее методы API построенной серверной части приложения (все POST-методы принимают на вход json, описанный в разделе 2.2; все без исключения методы возвращают json такого же формата; кроме того, все методы всегда возвращают код ответа 200 протокола HTTP, «сигнализируя» о неверном запросе через тело ответа):

а) /user/enter – POST-метод (*enterSystem* класса SystemClientFacade), возвращающий проекты пользователя с указанным от клиента e-mail;
б) /project/create – POST-метод (*createProject* класса ProjectFacade), позволяющий клиенту создать проект и возвращающий вновь созданный проект;
в) /project/edit – POST-метод (*editProject* класса ProjectFacade), позволяющий клиенту отредактировать проект (активировать/деактивировать его и/или изменить название); возвращает отредактированный в соответствии с запросом проект;
г) /project/check – POST-метод (*runModelChecking* класса ProjectFacade), позволяющий по присланным клиентам атомарным предикатом, свойству, модели, а также идентификатору проекта провести верификацию; возвращает то же, что прислал клиент, если проверка запущена, и сообщение с особым телом, если проверка для данного пользователя не может состояться;
д) /project/get/{id} – GET-метод (*getProject* класса ProjectFacade), отдающий информацию о проекте без истории верификаций над проектом и информации о его владельце;

е) /project/get/history/{id} – GET-метод (*getHistory* класса ProjectFacade), возвращающий информацию об истории проверки на моделях проекта с указанным id.

2.4.3. Развертывание приложения. Для развертывания серверной части приложения на сервере приложений WildFly 10 на машине с предустановленными JDK 1.8 и PostgreSQL 9.5 были выполнены следующие действия:

а) вручную добавлен и сконфигурирован в качестве одного из DataSouces JDBC-драйвер PostgreSQL на сервере приложений WildFly 10 (рекомендованные на официальном сайте JBoss установка и конфигурирование JDBC-драйвера через Web-интерфейс администратора WildFly не дали результата);

б) вручную добавлен и сконфигурирован в качестве используемого модуля jar-файл имплементации EclipseLink, позволяющий использовать object-relational mapping при работе приложения;

в) через Web-интерфейс администратора WildFly 10 произведено добавление и развертывания серверной части приложения.

2.5. Транспортная модель приложения. В этом параграфе опишем модель, использующуюся для обмена данными сервером и клиентами.

Как говорилось в параграфе 2.2, разумно определение транспортной модели на клиенте. Приведем диаграмму классов транспортной модели (рисунок 20):

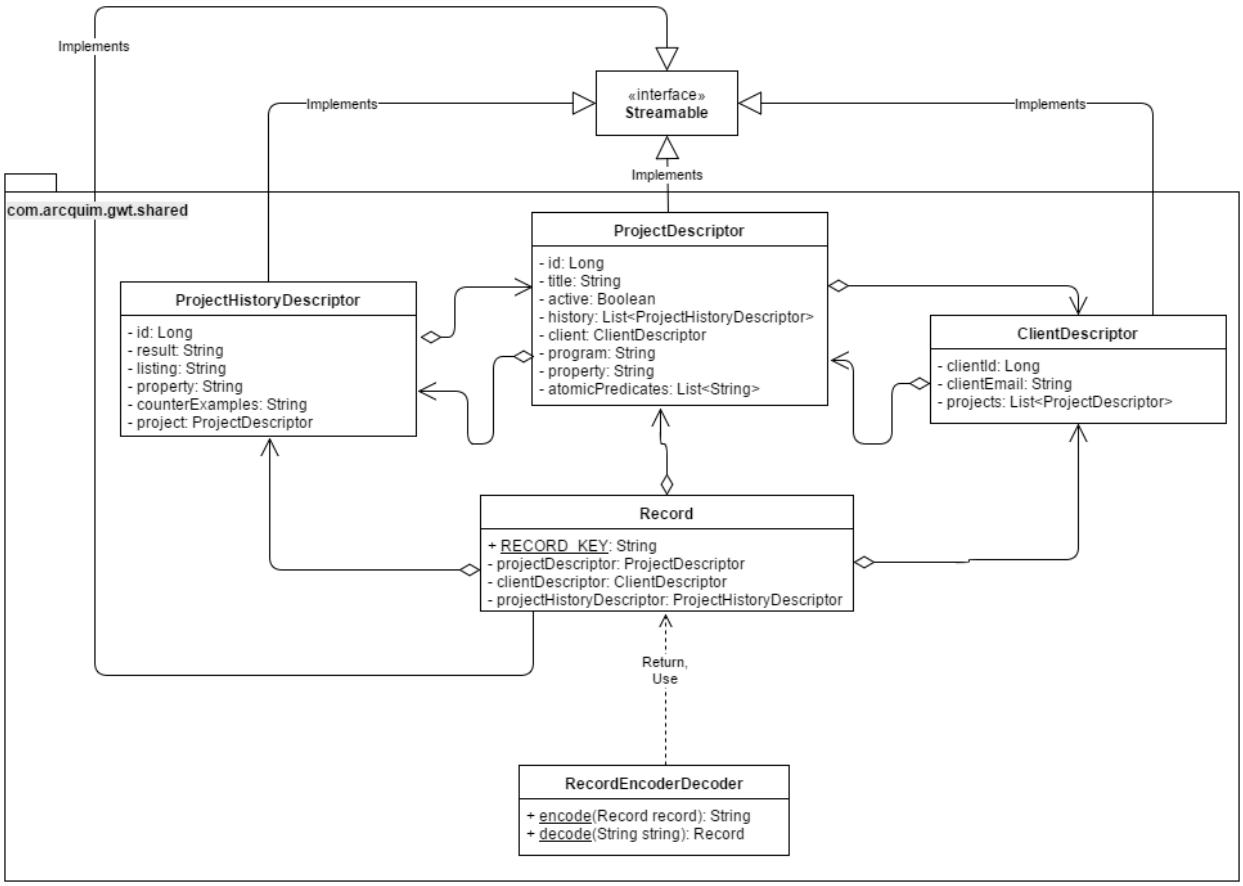

Рисунок 20. Диаграмма классов транспортной модели

Благодаря зависимости серверного модуля на клиентский модуль, написанный на GWT, а так же и благодаря тому, что код, написанный с помощью фреймворка GWT, можно интерпретировать и как Java-код, и как JavaScript-код, получаемый трансляцией Java-кода, транспортная модель едина для серверной части и клиентской gwt-части. Следует отметить, что здесь все классы, за исключением утилитарного класс RecordEncoderDecoder, реализуют маркировочный интерфейс Streamable, расположенный в библиотеке gwt-streamer. Класс RecordEncoderDecoder использует в своей работе singletone Streamer той же библиотеки, позволяющий элементарным путем проводить преобразование строки в объект класса Record и объекта класса Record в строку.

Далее приведем диаграмму классов серверной части, относящейся к транспортной модели (рисунок 21):

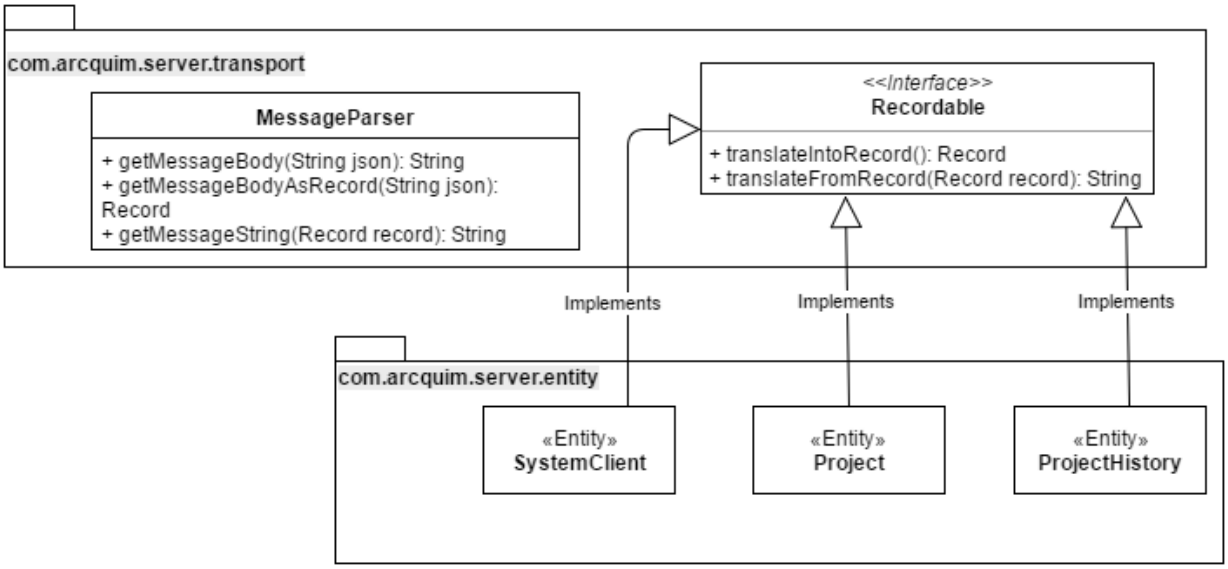

Рисунок 21. Диаграмма классов сервера, относящихся к транспортной модели

Здесь все классы сущности, за исключением Result, реализуют интерфейс Recordable. Таким образом, на уровне классов-сущностей определяется, как переводить серверные данные в данные для клиента и как данные от клиента переводить в серверные данные. Кроме того, полученный от объекта класса-сущности объект Record соответствующий EJB-бин (см. п. 2.4.2) может определенным образом изменить. Класс MessageParser является утилитарным, работая с json-строкой, описанной в п. 2.2.

2.6. GWT-модуль приложения. В этом параграфе приведем описание Web-клиента приложения, написанного с помощью фреймворка GWT.

2.6.1. Анализ задачи. Исходя из REST API серверной части приложения, описанного в п. 2.4.2, определим, какие страницы должно предоставить клиентское приложение пользователю.

Во-первых, работа с приложением начинается с того, что пользователю необходимо ввести адрес электронной почты – именно так он может получить доступ к проектам, для которых сможет выполнить верификацию. Поэтому стартовой страницей приложения логично сделать такую страницу, на которой пользователь смог бы ввести e-mail и, нажав на кнопку, перейти к странице с проектами.

Во-вторых, проектов у пользователя может не быть: например, он в первый раз зашел в систему. Поэтому необходимо на странице проектов отобразить соответствующее сообщение, если проектов у пользователя нет. Если же они есть, логично отобразить их в виде таблицы, где в каждом ряду отображать название проекта, активен он или нет, а также элементы управления, позволяющие просмотреть историю проекта и выполнить на нем верификацию – например, кнопки. Можно сделать также редактирование проекта – изменение его название и активности – прямо в таблице. В частности, компоненты CellTable и DataGrid GWT позволяют сделать это [9]. Также эти компоненты позволяют

сделать динамическое добавление и удаление строк, что открывает возможность организовать функционал создания проекта на странице проектов.

В-третьих, для верификации необходимо предоставить пользователю возможность вводить атомарные предикаты, свойство и модель. Можно сделать это на одном экране. При этом пользователь должен сначала ввести атомарные предикаты, а потом свойство, базирующееся на них. Соответствие введенным предикатам их индексам можно отобразить также с помощью CellTable или DataGrid GWT.

В-четвертых, историю проекта логично отобразить в виде таблицы, для чего аналогично можно использовать CellTable или DataGrid GWT.

Далее, следуя общепринятому мнению, что GWT отлично подходит именно для построения Single Page Application (SPA) [9], остановимся именно на этом подходе к организации пользовательского интерфейса. В этом случае необходимо предусмотреть элементы управления, позволяющие навигироваться по страницам приложения (например, кнопка Back).

Наконец, вопрос со стилизацией можно решить, используя библиотеку GWT-Bootstrap, позволяющую использовать стили Twitter в компонентах GWT [15]. Кроме того, эта библиотека позволит решить проблемы с показом пользователю различных сообщений.

2.6.2. Реализация. Для общения клиента с REST API, предоставляемого сервером, был выбран стандартный механизм асинхронных запросов GWT, инкапсулированный в стандартном классе RequestBuilder. Приведем диаграмму классов части клиентского приложения, отвечающего за обмен сообщениями с сервером (рисунок 22):

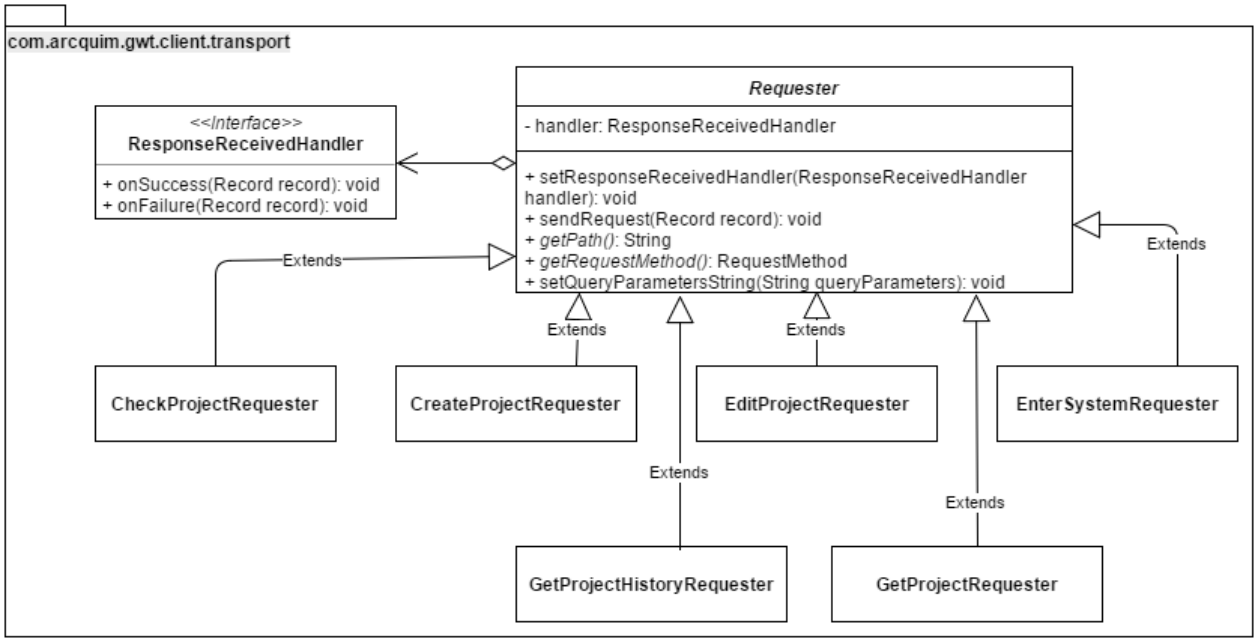

Рисунок 22. Диаграмма классов клиента, отвечающих за общение с сервером

Здесь абстрактный класс Requester инкапсулирует в себе логику отправки запроса с помощью RequestBuilder; после же получения ответа или в случае неудачи в методе *sendRequest* вызывается метод *onSuccess* или *onFailure* callback-а, являющегося реализацией интерфейса ResponseReceivedHandler (о том, какие классы реализуют этот интерфейс, речь пойдет ниже). Наконец, для каждого метода REST API сервера есть свой наследник Requester, содержащий информацию о том, по какому адресу и с помощью какого HTTP-метода можно обратиться к серверу.

Далее рассмотрим, как реализована навигация в приложении (рисунок 23):

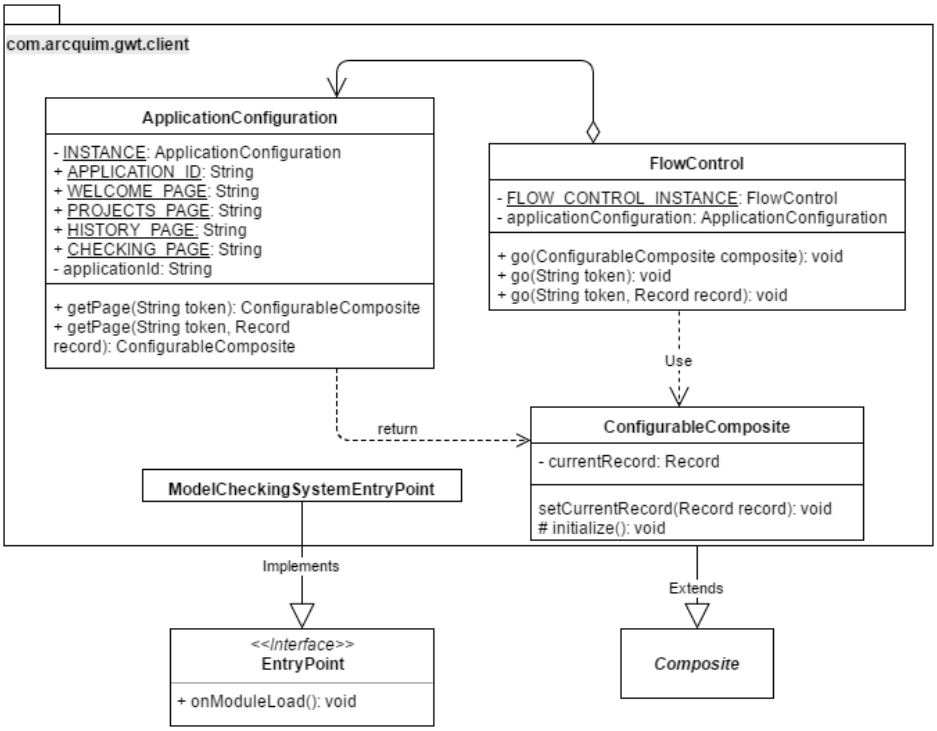

Рисунок 23. Диаграмма классов, отвечающих за навигацию.

Здесь изображена диаграмма классов пакета com.arcquim.gwt.client. Процесс навигации устроен следующим образом: каждая страница SPA наследует класс ConfigurableComposite, храня токен – строку из тех, что определены в ApplicationConfiguration. Для перехода со страницы на страницу (то есть с одного наследника ConfigurableComposite к другому) используется метод *go* FlowControl, который, получая ConfigurableComposite по токену от объекта ApplicationConfiguration, ставит его в элемент на HTML-странице с *id*, равным *applicationId* из ApplicationConfiguration (в данном случае это элемент *div*). При старте приложения происходит автоматическое отображение наследника ConfigurableComposite с токеном, совпадающим с WELCOME_PAGE.

Наконец, на следующей диаграмме классов представим всю систему, опустив некоторые утилитарные классы (рисунок 24):

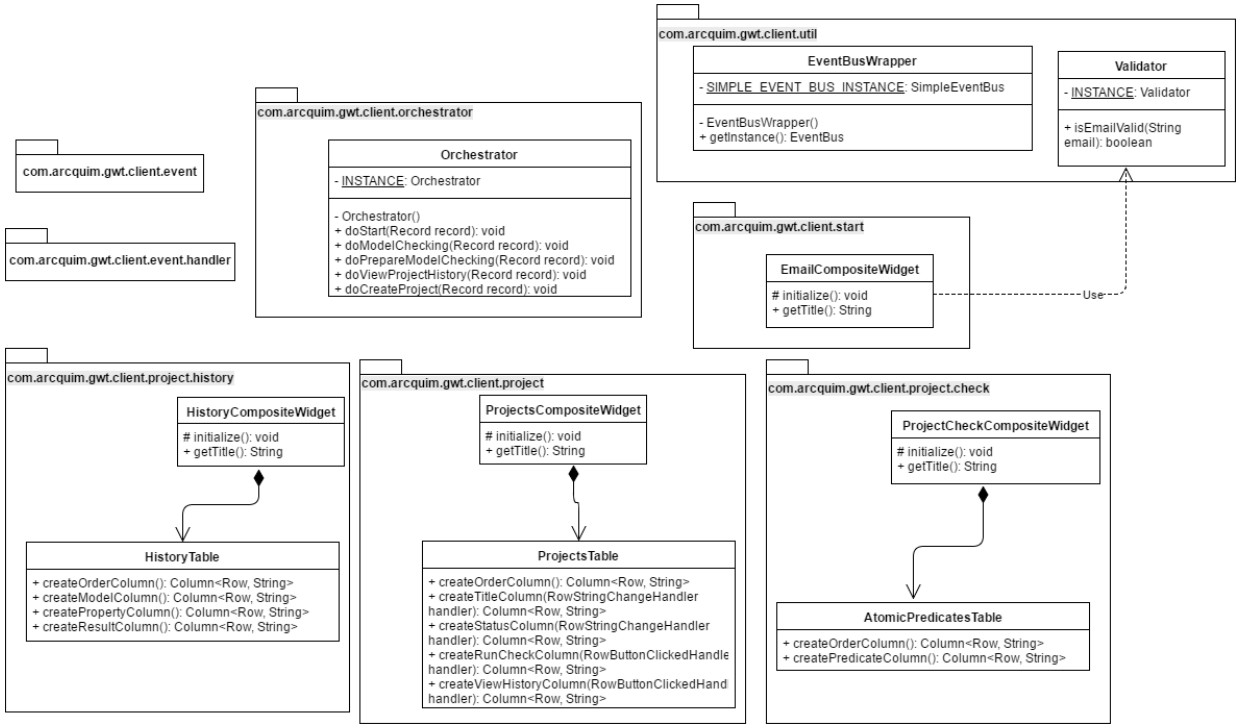

Рисунок 24. Диаграмма классов Web-клиента

Каждый класс, оканчивающийся на CompositeWidget, является наследником ConfigurableComposite. При определенном событии (например, нажатие на соответствующую кнопку) объект такого класса создает некоторое событие (наследник стандартного класса GWT Event) и через обертку над шиной событий (EventBusWrapper) «поджигает» его. У каждого события определен его тип, а также обработчик, который ставится к событию этого типа либо объектом класса Orchestrator, либо самим наследником ConfigurableComposite. На этой диаграмме опущены классы событий системы, интерфейсы их обработчиков, а также реализации этих интерфейсов. Кроме того, каждая таблица клиента наследует кастомизированного наследника стандартного компонента GWT CellTable, который определяет методы добавление и удаления строк, а также создания редактируемых ячеек.

Также следует упомянуть, что CSS для компонентов данного приложения созданы с помощью стандартного механизма GWT: создания наследника интерфейсов ClientBundle и CssResource.

2.6.3. Интерфейс. В этом пункте приведем описание интерфейса Web-клиента, написанного на GWT.

Первой страницей клиента является страница, где пользователь может ввести адрес электронной почты (рисунок 25):

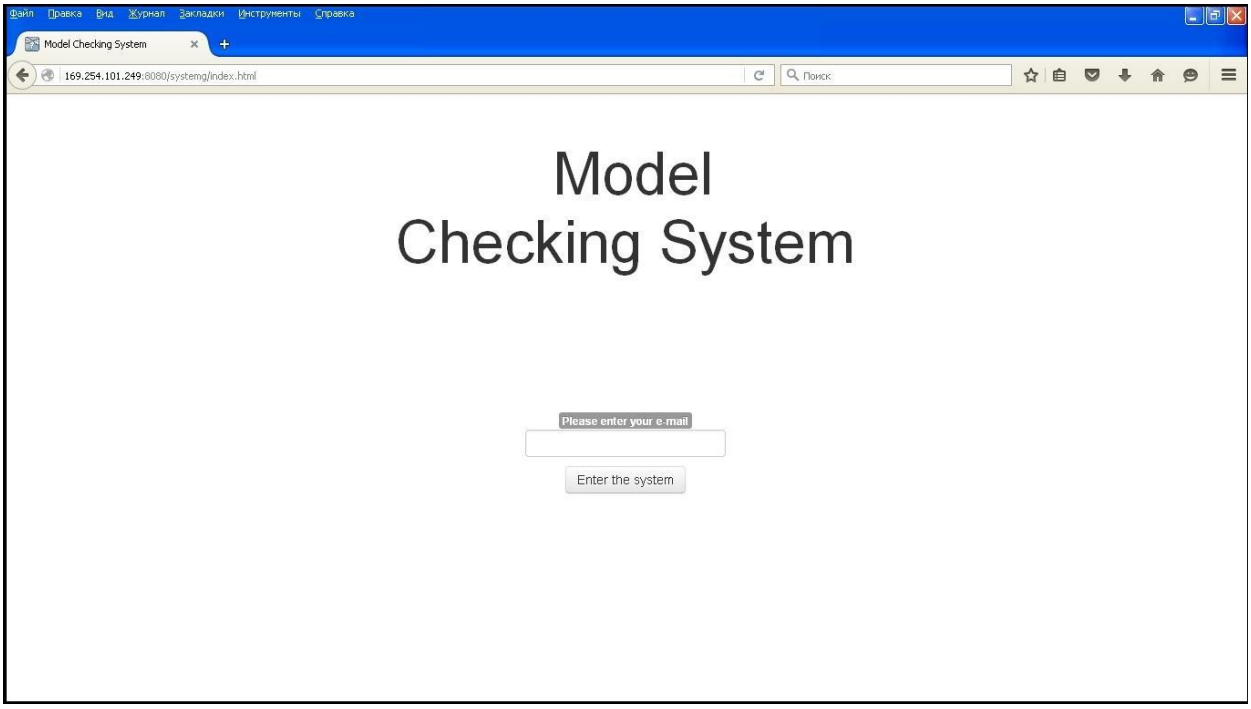

Рисунок 25. Главная страница Web-клиента

Пусть пользователь вводит невалидный e-mail и жмет на кнопку «Enter the system». В таком случае он увидит сообщение об ошибке (рисунок 26):

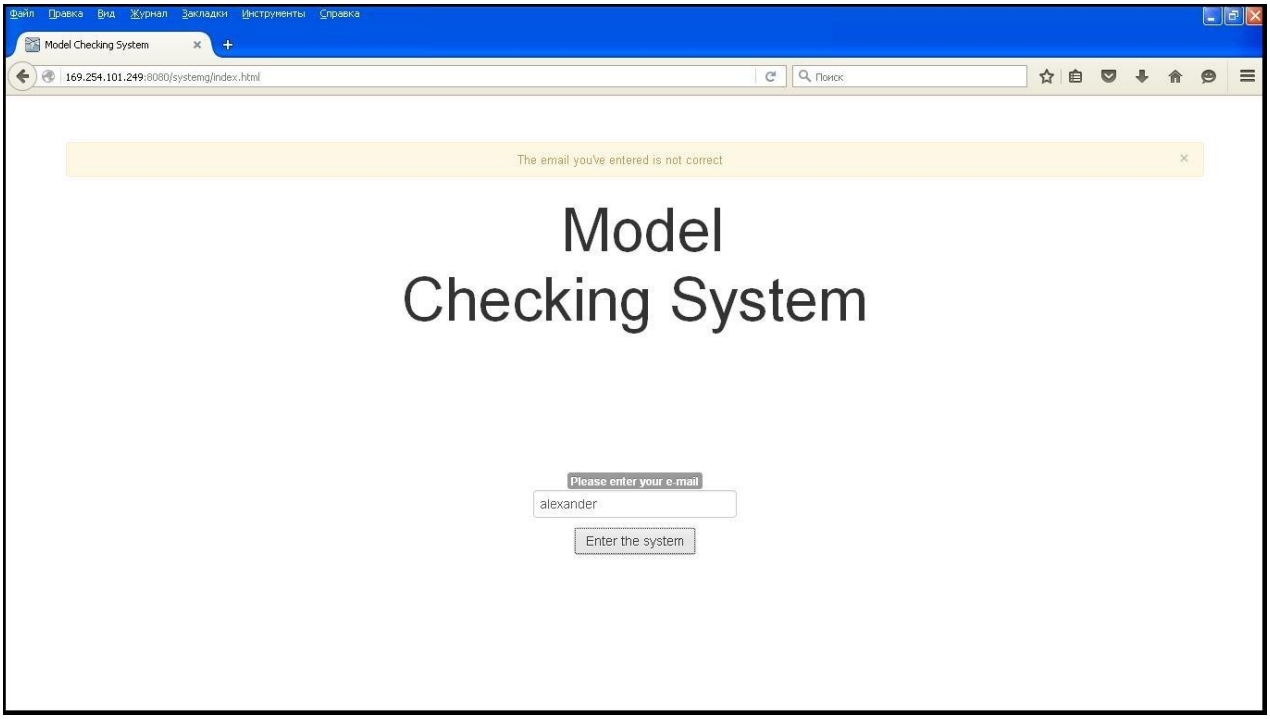

Рисунок 26. Сообщение о невалидном e-mail

Теперь пусть пользователь вводит корректный e-mail, после чего нажимает на кнопку «Enter the system». Пусть также у этого пользователя пока нет созданных проектов. В этом случае пользователь видит следующую страницу (рисунок 27):

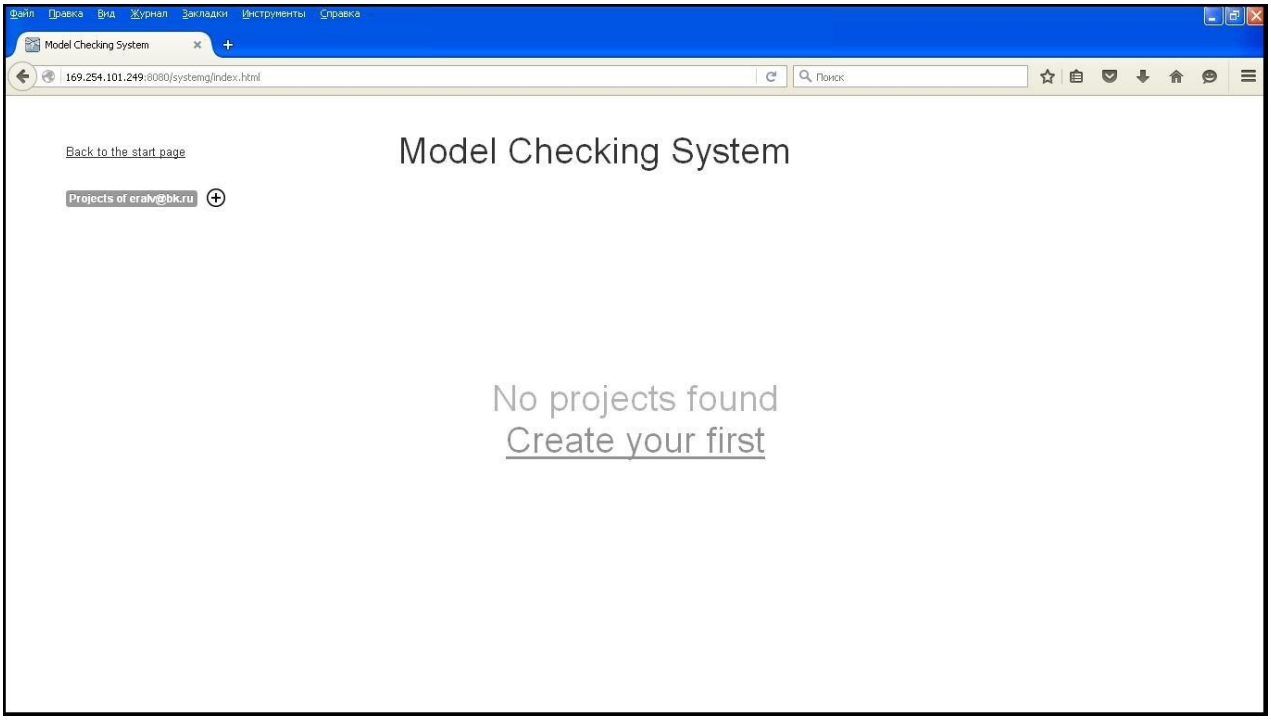

Рисунок 27. Страница проектов

Далее пользователь создает проект (рисунок 28):

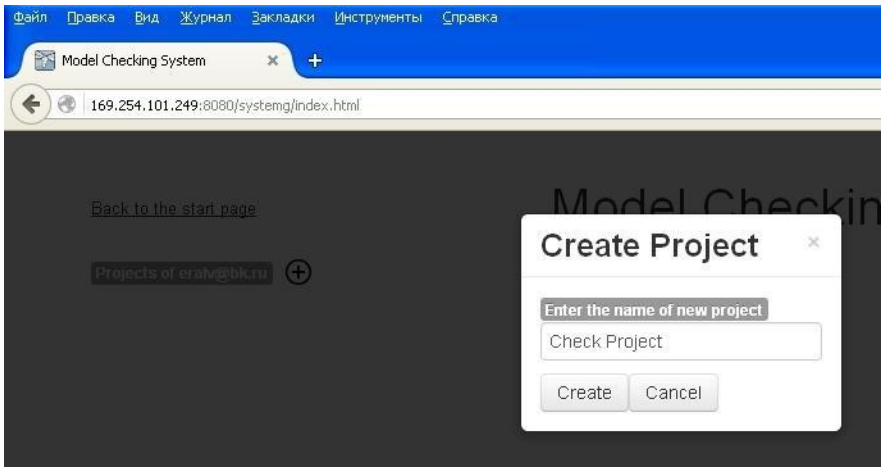

Рисунок 28. Создание проекта

После нажатия на кнопку «Create» popup исчезает и пользователь видит, что среди его проектов появился новый (рисунок 29):

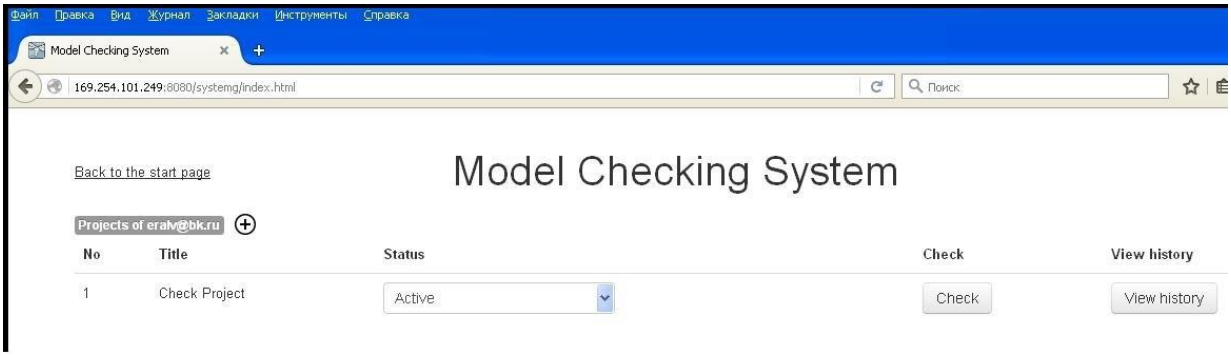

Рисунок 29. Страница проектов

Теперь пользователь может изменить название проекта (рисунок 30):

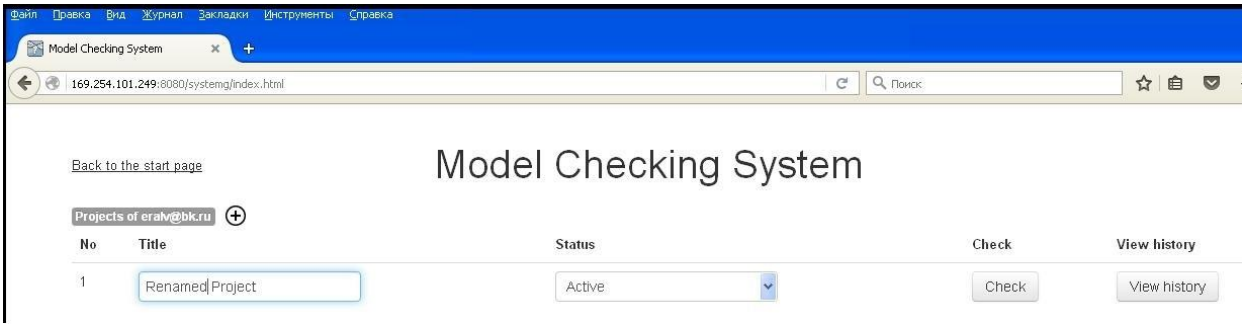

Рисунок 30. Переименование проекта

Теперь пользователь деактивирует проект (рисунок 31):

Рисунок 31. Деактивация проекта

Пытаясь выполнить верификацию над неактивным проектом, пользователь получает следующее сообщение об ошибке (рисунок 32):

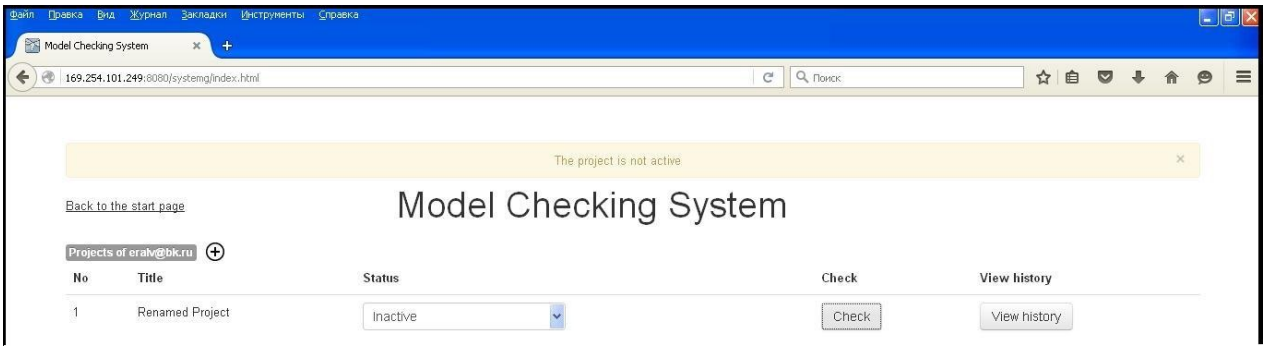

Рисунок 32. Сообщение о неактивности проекта

Далее пользователь делает проект активным, жмет на кнопку «Check» и попадает на страницу проверки модели этого проекта (рисунок 33):

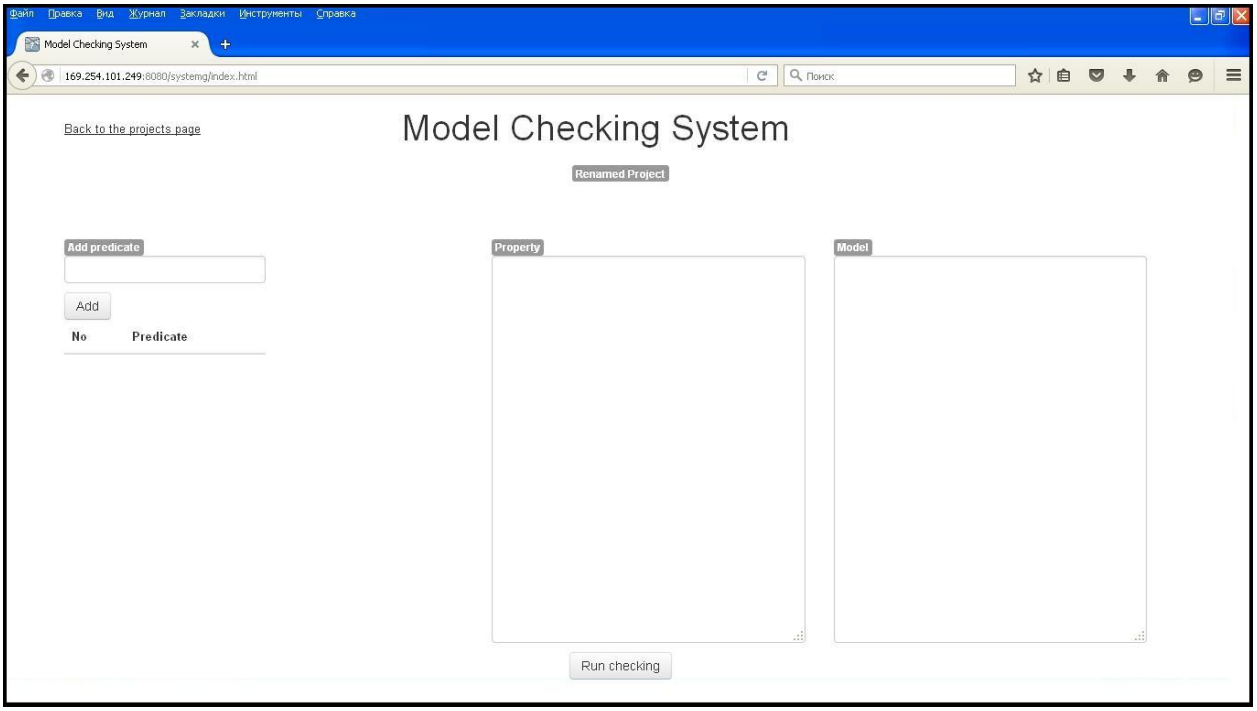

Рисунок 33. Страница ввода модели и свойства

Здесь пользователь вводит атомарные предикаты, задает свойство и описывает модель, после чего жмет на кнопку «Run checking» (рисунок 34):

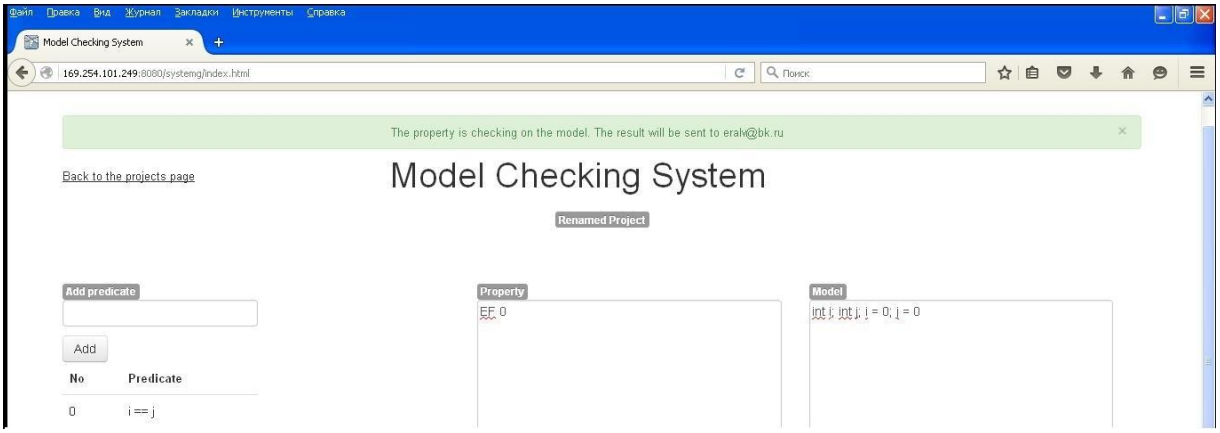

Рисунок 34. Сообщение о начале верификации

Сообщение о том, что верификация начата, а результат будет отправлен на соответствующий e-mail, отображается пользователю. Повторная попытка запустить верификацию приводит к сообщению об ошибке (рисунок 35):

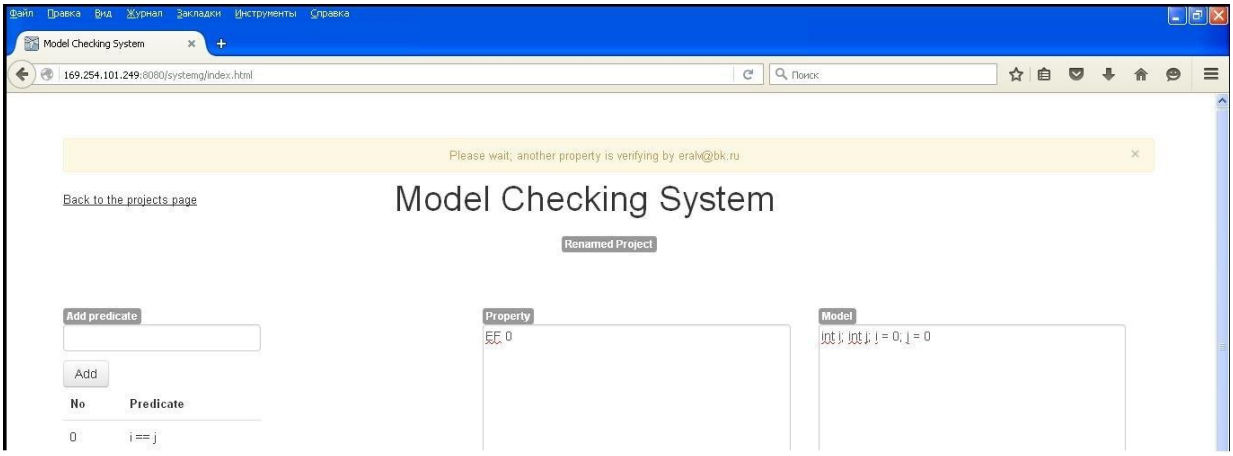

Рисунок 35. Сообщение о невозможности верификации

Пока верификация не выполнена, история проекта пуста (рисунок 36):

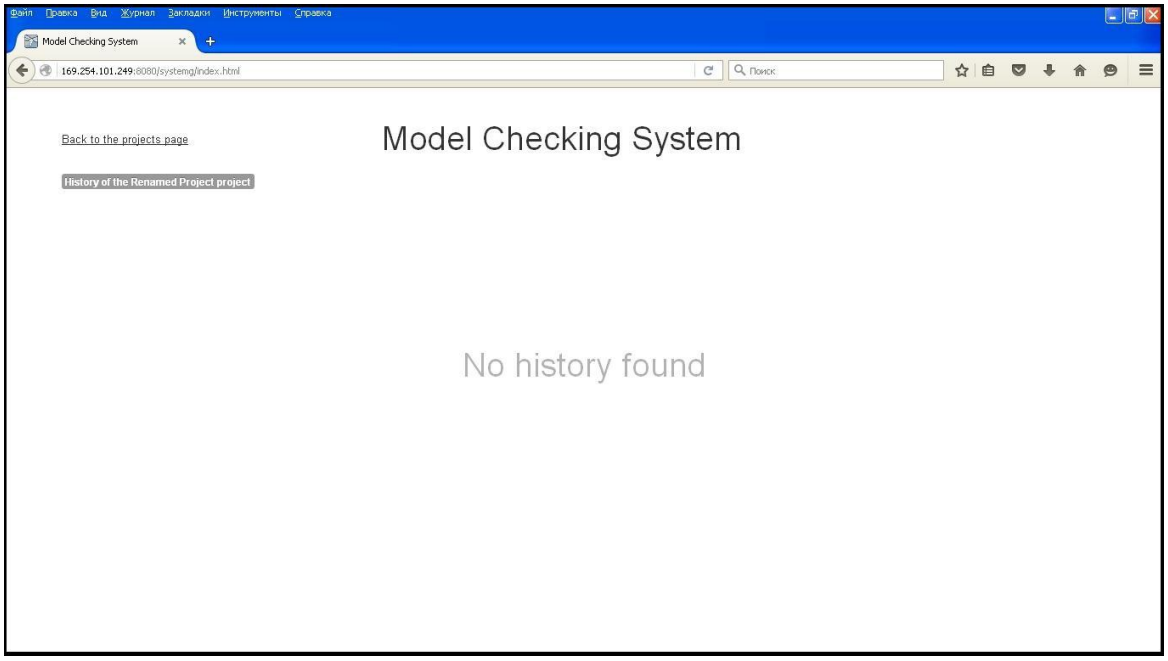

Рисунок 36. Страница истории проекта

Наконец, пользователю приходит письмо о результате верификации (рисунок 37):

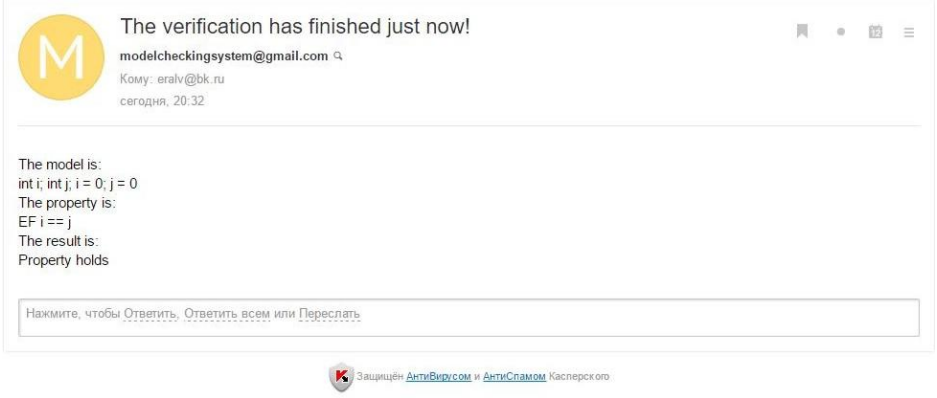

Рисунок 37. Письмо о завершении верификации

После этого на странице истории проекта можно увидеть запись (рисунок 38):

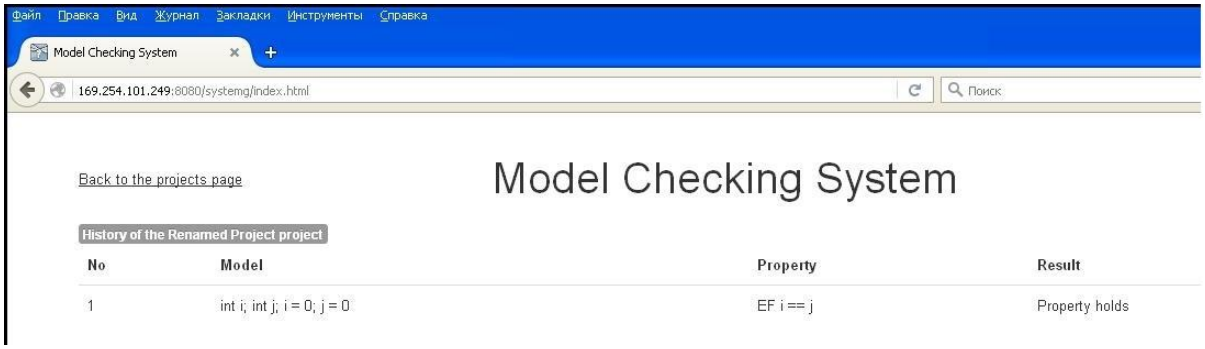

Рисунок 38. Страница истории проекта

2.7. Android-клиент приложения. В этом параграфе приведем описание Android-клиента приложения.

2.7.1. Анализ задачи. Необходимо отметить, что построение Android-клиента для приложения по количеству и функциональности экранов не должно сильно отличаться от страниц GWT-клиента (см. п. 2.6.1). Единственным исключением является метод задания атомарных предикатов, свойства и модели. Логично сделать эти три составляющие на разных экранах. В качестве минимальной поддерживаемой версии Android API остановимся на 14.

2.7.2. Реализация. Реализация Android-клиента по архитектуре и подходу практически аналогична реализации GWT-клиента; здесь, однако, транспортная модель, описанная в п. 2.5, внесена простым копированием. Часть, отвечающая за общение с сервером, также аналогична той, что описана в п. 2.6.2 и представлена на рисунке 22. Единственным исключением является то, что обращение к серверу здесь организовано с помощью стандартного класса Java HttpUrlConnection. Наконец, здесь элементарные компоненты пользовательского интерфейса – так называемые activities, наследники класса Android API AppCompatActivity – обращаются к объекту класса Orchestrator напрямую, не через механизм событий. Также здесь появляется дополнительный класс RequesterFactory, объект которого по вызову объекта класса Orchestrator отдает последнему определенного наследника абстрактного класс Requester (см. п. 2.6.2, рисунок 22).

2.7.3. Интерфейс. В этом пункте приведем описание интерфейса Android-клиента приложения.

При старте приложения пользователь видит следующий экран (рисунок 39):

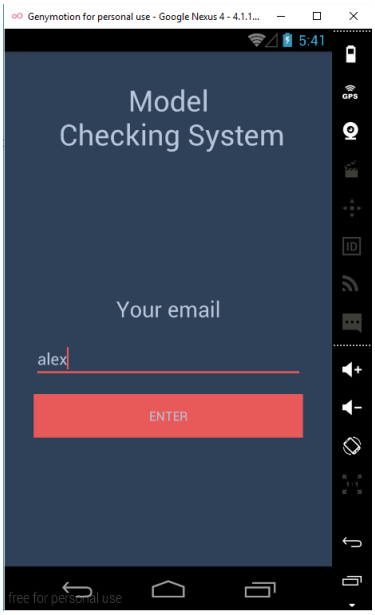

Рисунок 39. Главный экран приложения

Здесь пользователь ввел невалидный e-mail и нажал на кнопку «Enter», в результате чего увидел предупреждение (рисунок 40):

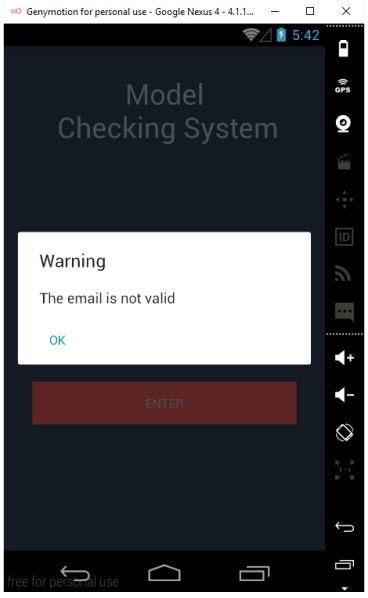

Рисунок 40. Сообщение о невалидном e-mail

Далее вводится валидный e-mail и пользователь переходит к своим проектам (рисунок 41):

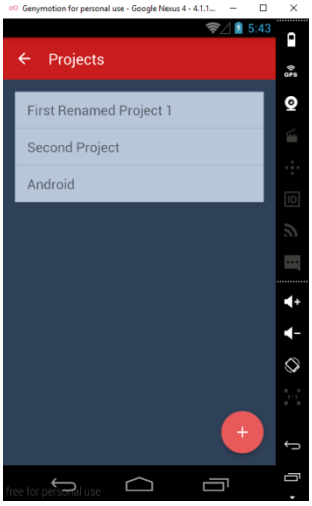

Рисунок 41. Экран проектов пользователя

Пользователь добавляет новый проект (рисунок 42):

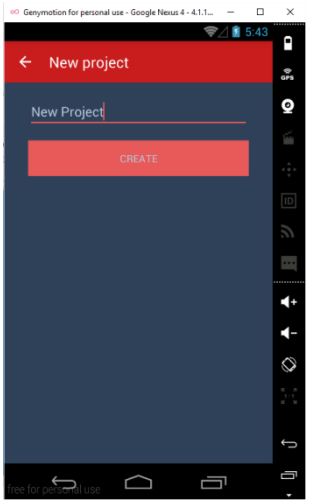

Рисунок 42. Экран создания проекта

После нажатия на кнопку «Create» происходит автоматический переход на экран с проектами (рисунок 43):

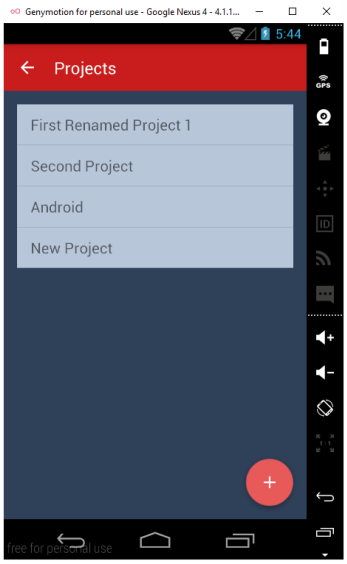

Рисунок 43. Экран проектов пользователя

Теперь пользователь решает зайти на страницу вновь созданного проекта (рисунок 44):

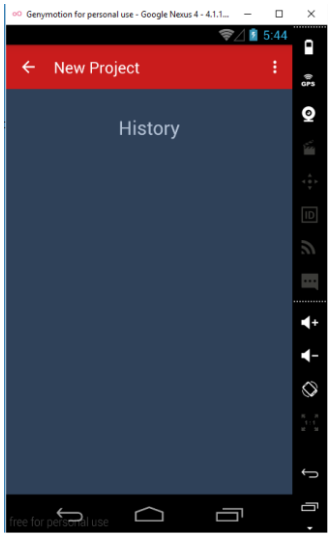

Рисунок 44. Экран истории проекта

Далее пользователь открывает контекстное меню (рисунок 45):

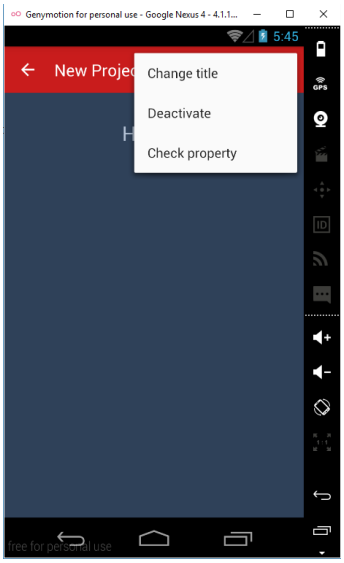

Рисунок 45. Контекстное меню страницы проекта

После он решает переименовать проект (рисунок 46):

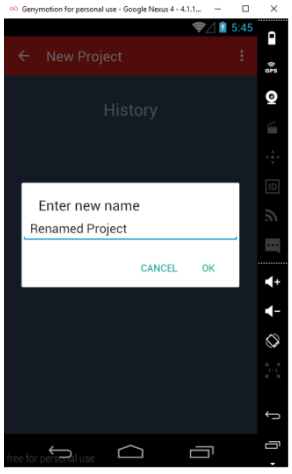

Рисунок 46. Переименование проекта

После переименования следует деактивация проекта (рисунок 47):

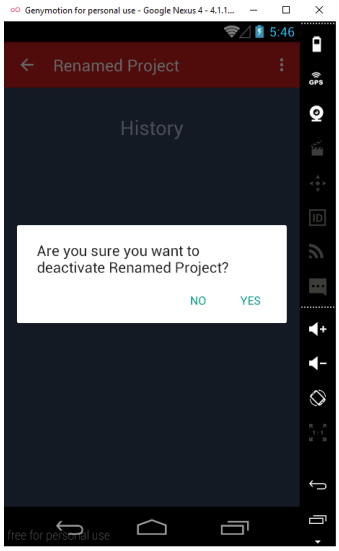

Рисунок 47. Деактивация проекта

Неактивный проект не может быть верифицирован; в связи с этим в контекстном меню такого проекта нет пункта «Check» (рисунок 48):

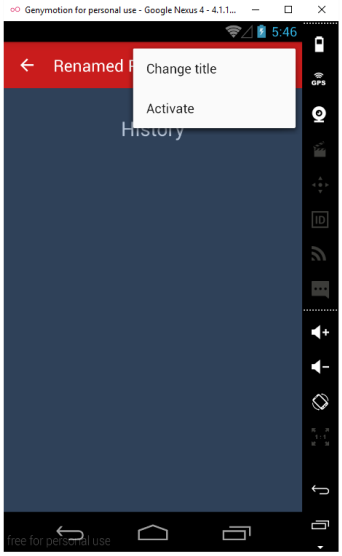

Рисунок 48. Контекстное меню неактивного проекта

Далее пользователь активирует проект и переходит к проверке модели: сначала он задает саму модель (рисунок 49):

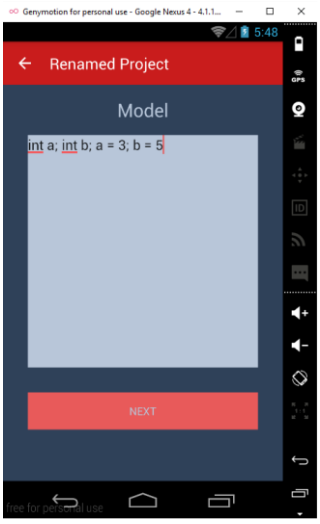

Рисунок 49. Экран ввода модели

После нажатия на «Next» можно ввести атомарные предикаты (рисунок 50):

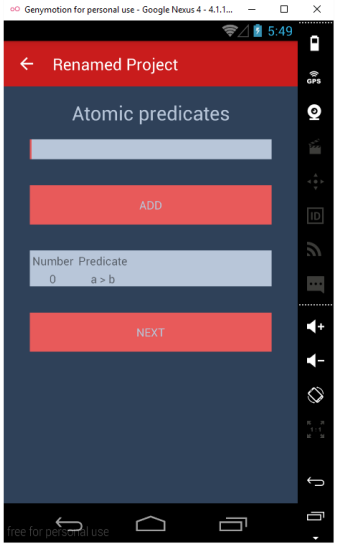

Рисунок 50. Экран ввода атомарных предикатов

После нажатия на «Next» переходим к вводу свойства (рисунок 51):

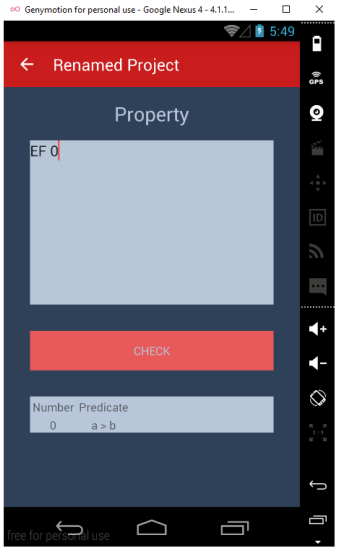

Рисунок 51. Экран ввода свойства

Нажатие на кнопку «Check» приводит к отправке на сервер введенных данных; верификация начинается, а пользователь видит следующее сообщение (рисунок 52):

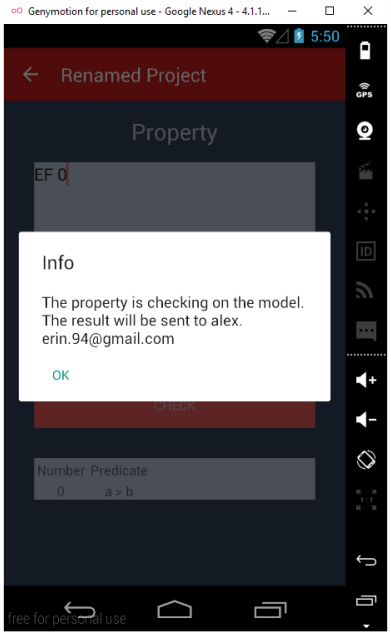

Рисунок 52. Сообщение о начале верификации проекта

Повторная отправка на верификацию приводит к следующему (рисунок 53):

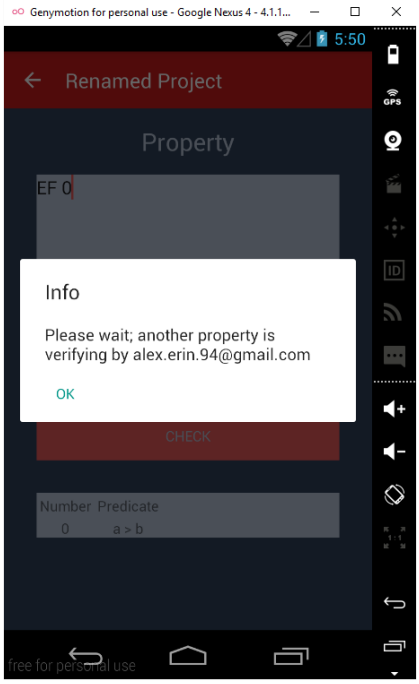

Рисунок 53. Сообщение о невозможности верификации проекта

После завершения верификации пользователю на электронную почту приходит соответствующие письмо (рисунок 54):

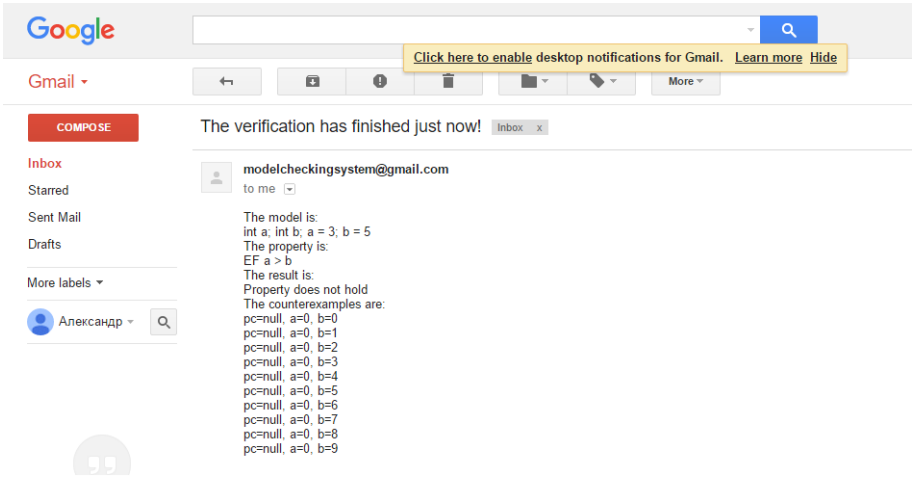

Рисунок 54. Письмо о завершении верификации

После этого на странице проекта пользователь видит наличие записи в истории (рисунок 55):

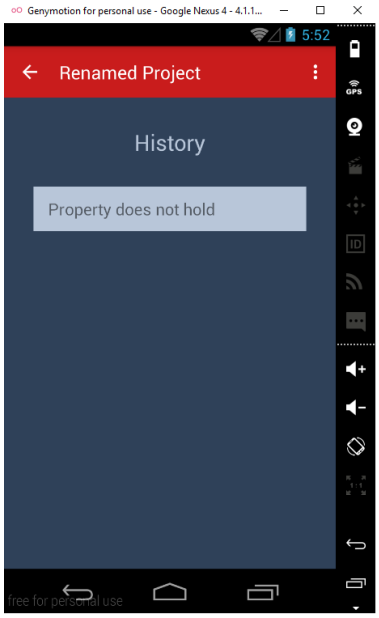

Рисунок 55. Экран истории верификации проекта

Нажатие на результат верификации приводит к появлению следующего экрана (рисунок 56):

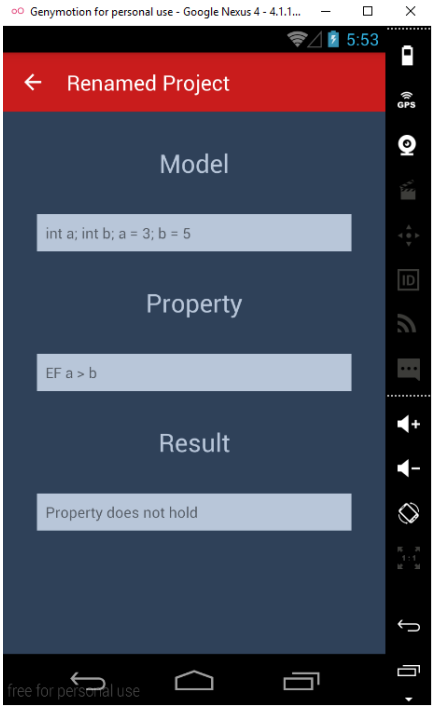

Рисунок 56. Экран верификации

2.8. Средства реализации. Перечислим программные средства, использовавшиеся для реализации приложения. Для построения четырехмодульного приложения, содержащего в себя такие модули, как реализация библиотеки для работы с OBDD, модуль-верификатор, серверная часть и клиентская на GWT использовался Java EE SDK 7.0; для реализации Android-клиента использовался Android SDK 23. В качестве сервера приложений выступал WildFly 10.

Использовались следующие среды разработки:

а) NetBeans 8.1 для разработки четырехмодульного приложения (библиотека работы с OBDD, верификатор, серверная часть приложения и клиентская часть на GWT);

б) Android Studio 1.5.1 для разработки Android-клиента приложения.

Также применялись следующие библиотеки и фреймворки:

а) Log4j версии 1.2.17 для логирования действий модуля-верификатора;

б) JUnit версии 4.10 для тестирования работы реализации библиотеки работы с OBDD и модуля-верификатора;

в) JavaBDD версии 1.0.1 как прототип для частично реализованной библиотеки работы с OBDD;

г) EclipseLink версии 2.5.2 как реализация стандарта JPA 2.1 для осуществления объектно-реляционного отображения (ORM) в рамках построения серверной части приложения;

д) Jboss VFC версии 3.2.11для автообнаружения Entity-классов на сервере приложений WildFly;

е) EJB версии 3.2 для транзакционного менеджмента и построения бизнес логики приложения (серверная часть), технология идет вместе с Java EE;
ж) JAX-RS версии 2.0.1 для построения REST API сервера приложения, технология включена в Java EE;
з) JavaMail API версии 1.5.5 для отправки электронных писем пользователям приложения;
и) JSON.simple версии 1.1.1 для парсинга и построения json-строк на серверной части приложения и на Android-клиенте;
к) GWT-streamer версии 2.0.0 для организации сериализации и десериализации объектов классов транспортной модели на сервере и обоих клиентах;
л) Google Web Toolkit (GWT) версии 2.7.0 для построения Web-клиента приложения;
м) GWT-Bootstrap версии 2.3.2 для стилизации компонентов Web-клиента.

Были применены следующие средства сборки:

а) Apache Maven версии 3.2 для сборки четырехмодульного проекта;
б) Gradle версии 2.13 для сборки Android-клиента.

Наконец, для тестирования всего приложения использовалась программа Oracle Virtual Box версии 5.0.12; для тестирования Android-клиента – Genymotion версии 2.6.0.

2.9. Требования к аппаратному и программному обеспечению. Приложение-сервер предназначено для запуска на Unix-подобных операционных системах и операционных системах семейства Windows (начиная с Windows Vista SP2); Web-клиент предназначен для работы в достаточно современных браузерах, работающих на движках Gecko, WebKit, Blink, Edge и Trident; Android-клиент требует для своей работы операционной системы Android версии 4.0 («Ice Cream Sandwich») и выше.

Требования к аппаратному обеспечению сервера:

а) процессор, реализующий архитектуру x86 или x64 (например, процессоры от Intel (начиная с Pentium II) и AMD (начиная с K6)) с тактовой частотой не менее 266 МГц;
б) объем свободной оперативной памяти не менее 1 Гб;
в) 1 Гб свободного дискового пространства для размещения приложения и ведения его логов; однако, следует отметить, что объем требуемого дискового пространства можно сократить, своевременно удаляя файлы с ненужными логами.

Требования к программному обеспечению сервера:

а) СУБД PostgreSQL версии 9.5 с настроенным расширением PL/Perl и выполненным скриптом создания базы данных (см. приложение 3);
б) Java EE 7 SDK;
в) настроенный в соответствии с п. 2.4.3 сервер приложений WildFly версии 10.

2.10. Тестирование. Тестирование модуля, представляющего собой частичную реализацию библиотеки работы с OBDD, тестировался с помощью Unit-тестов; пример такого теста приведен в приложении 7. Модуль-верификатор также тестировался путем написания Unit-тестов; пример такого теста также приведен в приложении 7. Для тестирования приложения в целом были созданы три виртуальные машины средствами Oracle Virtual Box и Genymotion. В частности, для развертывания приложения применялась виртуальная машина с операционной системой Ubuntu 16.04, к которой по соединению типа «локальная сеть» была подключена виртуальная машина с Windows XP; также был произведен проброс порта 8080 виртуальной машины с развернутым сервером на порт 2231 хоста. Android-клиент тестировался с помощью Genymotion, причем в качестве сервера для этого клиента выступала хост-машина; Android-клиент отправлял ей запросы на порт 2231.

ЗАКЛЮЧЕНИЕ

В результате данной работы:

а) изучены алгоритмы проверки моделей, подробно был исследован символьный алгоритм, основанный на использовании OBDD;
б) изучены базовые алгоритмы работы с OBDD, начата реализация библиотеки работы с OBDD (в частности, реализованы логические и предикаторные операции);
в) обнаружены некоторые особенности работы JVM на операционных системах семейства Windows;
г) создано многомодульное клиент-серверное приложение, предназначенное для символьной проверки моделей.

Реализация вышеназванного клиент-серверного приложения включила в себя:

а) проектирование и создание модуля-верификатора, выполняющего по модели, представленной на формальном C-подобном языке, набору атомарных предикатов и свойству символьную верификацию, основанную на двоичных разрешающих диаграммах, выполнимости заданного свойства на заданной модели (что включает в себя создание компилятора, переводящего программу на формальном языке в логическую формулу);
б) использование самостоятельно написанной библиотеки для работы с OBDD, получение подтверждения корректности работы написанной библиотеки;
в) проектирование и реализацию серверной части приложения, что включило в себя создание небольшой базы данных и написание REST API;
г) проектирование и реализацию Web-клиента приложения как Single Page Application с помощью GWT;
д) проектирование и реализацию Android-клиента приложения.

Кроме того, в процессе реализации приложения был достаточно подробно изучен сервер приложений WildFly. В дальнейшем планируется расширение приложения: написание части, обеспечивающей разграничение пользователей по ролям, а также поддержка темпоральной логики линейного времени (LTL). Кроме того, планируется реализация возможности изменения порядка булевских переменных в библиотеке для работы с OBDD.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Верификация автоматных программ / С.Э. Вельдер [и др.]. – СПб. : Наука, 2011. – 244 с.
2. Кларк М.Э. Верификация моделей программ: Model Checking / М.Э. Кларк, О. Грамберг, Д. Пелед. – М. : Издательство Московского центра непрерывного математического образования, 2002. – 416 с.
3. List of model checking tools. – URL: https://en.wikipedia.org/wiki/List_of_model_checking_tools (дата последнего обращения: 29.05.2016).
4. Карпов Ю.Г. Model Checking. Верификация параллельных и распределенных программных систем / Ю.Г. Карпов – СПб. : БХВ-Петербург, 2010. – 551с.
5. Drechsler R. Binary Decision Diagrams: Theory and Implementation / R.Drechsler, B. Becker. – New York, USA: Springer, 1998. – 200 с.
6. Wegener I. Branching Programs and Binary Decision Diagrams: Theory and Implementation / I.Wegener. – Philadelphia, USA: Society for Industrial and Applied Mathematics, 200. – 408 с.
7. Guide to Working with Multiple Modules. – URL: https://maven.apache.org/guides/mini/guide-multiple-modules.html (дата последнего обращения: 30.05.2016).
8. Building a Real Application with Maven. – URL: https://docs.oracle.com/middleware/1212/core/MAVEN/real_app.htm (дата последнего обращения: 30.05.2016).
9. Tacy A. GWT in Action: Second Edition / A. Tacy, R.Hanson, J. Essington. – New York, USA: Manning Publications Co. – 680 с.
10. Gwt-streamer: Google Code Archive – Long-term Storage for Google Code Project Hosting. – URL: https://code.google.com/archive/p/gwt-streamer/ (дата последнего обращения: 30.05.2016).
11. Asynchronous Method Invocation – The Java EE 6 Tutorial. – URL: https://docs.oracle.com/javaee/6/tutorial/doc/gkkqg.html (дата последнего обращения: 31.05.2016).
12. Container-Managed Transactions – The Java EE 5 Tutorial. – URL: http://docs.oracle.com/javaee/5/tutorial/doc/bncij.html (дата последнего обращения: 31.05.2016).
13. Email address parsing: PostgreSQL Wiki. – URL: https://wiki.postgresql.org/wiki/Email_address_parsing (дата последнего обращения: 31.05.2016).

14. Citext: PostgreSQL 9.5.3 Documentation. – URL: https://www.postgresql.org/docs/9.5/static/citext.html (дата последнего обращения: 31.05.2016).

15. Gwt-Bootstrap. – URL: https://gwtbootstrap.github.io/ (дата последнего обращения: 01.06.2016).

ПРИЛОЖЕНИЯ

Так как разработанное приложение достаточно громоздко, здесь приводится лишь небольшая часть исходного кода, на наш взгляд, ключевые моменты. Полный код приложения доступен в публичном репозитории на https://github.com/arcquim/CustomModelCheckingSystem.

Приложение 1. Часть исходного кода класса CustomBDD.

```
public class CustomBDD extends BDD implements Cloneable {

    //table that stores the OBDD
    private final Map<Integer, CustomBDDItem> bddMap;
    private Integer var;

    /*
    *
    * @return index of the boolean variable that is a head of the OBDD
    */
    @Override
    public int var() {
        if (var == null) {
            Map<Integer, Integer> variablesOrder = factory.getVariablesOrderMapping();
            Integer minimalOrder = factory.varNum(), requiredVar = -1;
            for (Integer variable : usedVariables.keySet()) {
                Integer variableOrder = variablesOrder.get(variable);
                if (variableOrder < minimalOrder) {
                    requiredVar = variable;
                    minimalOrder = variableOrder;
                }
            }
            var = requiredVar;
        }
        return var;
    }

     /*
    *
    * @return CustomBDD representing the result of the not operator over this CustomBDD
    */
    @Override
    public BDD not() {
        //creating table for new OBDD
        Map<Integer, CustomBDDItem> mapForNewBDD = new HashMap<>(bddMap);
        CustomBDDItem zeroItem = mapForNewBDD.get(CustomBDDItem.DEFAULT_ZERO_ITEM_KEY);
        CustomBDDItem oneItem = mapForNewBDD.get(CustomBDDItem.DEFAULT_ONE_ITEM_KEY);
        if (zeroItem != null && oneItem != null) { //the BDD is not empty
            //start of terminal variables parents swapping
            CustomBDDItem newZeroItem = new CustomBDDItem(zeroItem);
            newZeroItem.setAdditionalParents(oneItem.getAdditionalParents());
            CustomBDDItem newOneItem = new CustomBDDItem(oneItem);
            newOneItem.setAdditionalParents(zeroItem.getAdditionalParents());
            mapForNewBDD.put(CustomBDDItem.DEFAULT_ZERO_ITEM_KEY, newZeroItem);
            mapForNewBDD.put(CustomBDDItem.DEFAULT_ONE_ITEM_KEY, newOneItem);
            //parent swapped; let's make those parents linking to new children
            List<Integer> newZeroAdditionalParents = newZeroItem.getAdditionalParents();
            if (newZeroAdditionalParents != null) {
                //processing parent of new zero (old one)
                for (Integer additionalParent : newZeroAdditionalParents) {
                    CustomBDDItem newItem = new
CustomBDDItem(mapForNewBDD.get(additionalParent));
                    if (newItem.getZeroLink().equals(CustomBDDItem.DEFAULT_ONE_ITEM_KEY)) {
                        if (newItem.getOneLink().equals(CustomBDDItem.DEFAULT_ZERO_ITEM_KEY)) {
                            newItem.setZeroLink(null);
                        }
```

```
                    else {
                        newItem.setZeroLink(CustomBDDItem.DEFAULT_ZERO_ITEM_KEY);
                    }
                }
                else {
                    if (newItem.getZeroLink().equals(CustomBDDItem.DEFAULT_ZERO_ITEM_KEY)) {
                        newItem.setOneLink(null);
                    }
                    else {
                        newItem.setOneLink(CustomBDDItem.DEFAULT_ZERO_ITEM_KEY);
                    }
                }
                mapForNewBDD.put(additionalParent, newItem);
            }
        }
        List<Integer> newOneAdditionalParents = newOneItem.getAdditionalParents();
        if (newOneAdditionalParents != null) {
            //processing parent of new one (old zero)
            for (Integer additionalParent : newOneAdditionalParents) {
                CustomBDDItem nextItem = mapForNewBDD.get(additionalParent);
                if (nextItem.getZeroLink() != null && nextItem.getOneLink() != null) {
                    nextItem = new CustomBDDItem(nextItem);
                    mapForNewBDD.put(additionalParent, nextItem);
                }
                if (CustomBDDItem.DEFAULT_ZERO_ITEM_KEY.equals(nextItem.getZeroLink())) {
                    nextItem.setZeroLink(CustomBDDItem.DEFAULT_ONE_ITEM_KEY);
                    if (nextItem.getOneLink() == null) {
                        nextItem.setOneLink(CustomBDDItem.DEFAULT_ZERO_ITEM_KEY);
                    }
                }
                else {
                    nextItem.setOneLink(CustomBDDItem.DEFAULT_ONE_ITEM_KEY);
                    if (nextItem.getZeroLink() == null) {
                        nextItem.setZeroLink(CustomBDDItem.DEFAULT_ZERO_ITEM_KEY);
                    }
                }
            }
        }
    }
    else {//the CustomBDD is constant; we just need to reverse this constant
        if (zeroItem != null) {
            return factory.one();
        }
        else {
            return factory.zero();
        }
    }
    return new CustomBDD(mapForNewBDD, factory);
}

/*
* This operation merges three OBDDs into one
* @param stepResultOne the then-OBDD for the head of the OBDD
* @param stepResultZero the else-OBDD for the head of the OBDD
* @return CustomBDD representing the result of the ifThenElse operator over
* this CustomBDD, then-CustomBDD and else-CustomBDD
*/
public CustomBDD ifThenElse(CustomBDD stepResultOne, CustomBDD stepResultZero) {
    Integer thisVar = this.var();
    CustomBDDItem newHeadItem = new CustomBDDItem(this.usedVariables.get(thisVar).get(0));
    Map<Integer, CustomBDDItem> items = new HashMap<>();

    //reducing arguments before merging, considering OBDDs
    //under the non-head vertexes reduced
    stepResultZero.reduceVar();
    stepResultOne.reduceVar();

    if (stepResultOne.equals(stepResultZero)) {
        //in this case we do not need merging
```

```
            return stepResultZero;
        }
        //before merging we have to make ids unique in the tables of the arguments
        Integer newHeadItemIndex = makeIdsUnique(stepResultZero, stepResultOne);

        //getting heads on the agruments
        Integer stepResultZeroVar = stepResultZero.var();
        Integer stepResultOneVar = stepResultOne.var();
        CustomBDDItem varItemZero, varItemOne;
        Integer newHeadItemZeroLink = CustomBDDItem.DEFAULT_NEXT_ITEM_KEY,
                newHeadItemOneLink = CustomBDDItem.DEFAULT_NEXT_ITEM_KEY;
        if (stepResultZeroVar >= 0) {//stepResultZero is not a constant
            varItemZero = stepResultZero.usedVariables.get(stepResultZeroVar).get(0);
            //getting the immediate child of the head
            CustomBDDItem childItem = stepResultZero.bddMap.get(varItemZero.getZeroLink());
            if (childItem == null || childItem.getParentLink() == null) {
                childItem = stepResultZero.bddMap.get(varItemZero.getOneLink());
            }
            if (childItem == null) {
                return new CustomBDD(factory);
            }
            if (childItem.getParentLink() == null) {//the child is a terminal vertex
                List<Integer> additionalParents = childItem.getAdditionalParents();
                if (additionalParents == null || additionalParents.isEmpty()) {
                    return new CustomBDD(factory);
                }
                //searching for an index of the head vertex of stepResultZero
                for (Integer parent : additionalParents) {
                    CustomBDDItem parentItem = stepResultZero.bddMap.get(parent);
                    if (varItemZero == parentItem) {
                        newHeadItemZeroLink = parent;
                        break;
                    }
                }
            }
            else {//the child is non-terminal
                newHeadItemZeroLink = childItem.getParentLink();
            }
            //the head of stepResultZero is linking to the head of this CustomBDD
            varItemZero.setParentLink(newHeadItemIndex);
        }
        else {//the stepResultZero is just a constant
            newHeadItemZeroLink = stepResultZero.bddMap.get(CustomBDDItem.DEFAULT_ZERO_ITEM_KEY)
== null ?
                    CustomBDDItem.DEFAULT_ONE_ITEM_KEY : CustomBDDItem.DEFAULT_ZERO_ITEM_KEY;
            varItemZero = stepResultZero.bddMap.get(newHeadItemZeroLink);
            varItemZero.addAdditionalParent(newHeadItemIndex);
        }

        if (stepResultOneVar >= 0) {//stepResultOne is not a constant
            varItemOne = stepResultOne.usedVariables.get(stepResultOneVar).get(0);
            //getting the immediate child of the head
            CustomBDDItem childItem = stepResultOne.bddMap.get(varItemOne.getZeroLink());
            if (childItem == null || childItem.getParentLink() == null) {
                childItem = stepResultOne.bddMap.get(varItemOne.getOneLink());
            }
            if (childItem == null) {
                return new CustomBDD(factory);
            }
            if (childItem.getParentLink() == null) {//the child is a terminal vertex
                List<Integer> additionalParents = childItem.getAdditionalParents();
                if (additionalParents == null || additionalParents.isEmpty()) {
                    return new CustomBDD(factory);
                }
                //searching for an index of the head vertex of stepResultOne
                for (Integer parent : additionalParents) {
                    CustomBDDItem parentItem = stepResultOne.bddMap.get(parent);
                    if (varItemOne == parentItem) {
                        newHeadItemOneLink = parent;
```

```
                        break;
                    }
                }
            }
            else {//the child is non-terminal
                newHeadItemOneLink = childItem.getParentLink();
            }
            //the head of stepResultOne is linking to the head of this CustomBDD
            varItemOne.setParentLink(newHeadItemIndex);
        }
        else {//the stepResultOne is just a constant
            newHeadItemOneLink = stepResultOne.bddMap.get(CustomBDDItem.DEFAULT_ONE_ITEM_KEY) ==
null ?
                    CustomBDDItem.DEFAULT_ZERO_ITEM_KEY : CustomBDDItem.DEFAULT_ONE_ITEM_KEY;
            varItemOne = stepResultOne.bddMap.get(newHeadItemOneLink);
            varItemOne.addAdditionalParent(newHeadItemIndex);
        }
        //making the head vertex of new CustomBDD linking to its children
        newHeadItem.setZeroLink(newHeadItemZeroLink);
        newHeadItem.setOneLink(newHeadItemOneLink);
        newHeadItem.setParentLink(null);
        newHeadItem.setAdditionalParents(null);
        stepResultZero.bddMap.put(newHeadItemIndex, newHeadItem);
        stepResultOne.bddMap.put(newHeadItemIndex, newHeadItem);
        items.putAll(stepResultZero.bddMap);
        //we need to merge stepResultOne and stepResultZero, because we should
        //have two terminal vertexes only
        stepResultOne.mergeIntoAnotherBddMap(stepResultZero.bddMap);
        items.putAll(stepResultOne.bddMap);
        return new CustomBDD(items, factory);
    }

    /**
     *
     * @param bdd the CustomBDD instance describing the variables to apply quantifier over
     * @return CustomBDD instance: the result of existentional quantifier applying
     */
    @Override
    public BDD exist(BDD bdd) {
        if (bdd.getFactory() != factory || !(bdd instanceof CustomBDD)) {
            throw new IllegalArgumentException("Operations allowed only on "
                    + "BDDs with the same factory and implementation classes.");
        }
        CustomBDD customBDD = (CustomBDD) bdd;
        CustomBDD existResult = this;
        //we are considering the argument as a set of variables
        Set<Integer> thisBddVariables = usedVariables.keySet();
        for (Integer bddVariable : customBDD.usedVariables.keySet()) {
            //applying existentional quantifier over this OBDD and next variable
            if (thisBddVariables.contains(bddVariable)) {
                CustomBDD thisZero = existResult.replaceVariableWithValue(bddVariable, false);
                CustomBDD thisOne = existResult.replaceVariableWithValue(bddVariable, true);
                existResult = (CustomBDD) thisZero.or(thisOne);
            }
        }
        if (existResult == this) {//there are no variables to make quantification
            return new CustomBDD(existResult);
        }
        else {
            return existResult;
        }
    }

    @Override
    public BDD apply(BDD bdd, BDDFactory.BDDOp bddop) {
        if (bdd.getFactory() != this.factory || !bdd.getClass().equals(CustomBDD.class)) {
            // we are supporting only and only CustomBDD instances, that have the same factory
object
            throw new IllegalArgumentException("Operations allowed only on "
```

```
                + "BDDs with the same factory and implementation classes.");
        }
        CustomBDD customBDD = (CustomBDD) bdd;
        if (BDDFactory.or.equals(bddop)) {
            //perform the OR operation
            CustomBDD resultBdd = performOr(customBDD);
            //reduce the head of the result; non-head vertexes have been reduced already
            resultBdd.reduceVar();
            orCount++;
            if (orCount == 100) {//limit of the OR cache
                orCount = 0;
                factory.currentOrCache.clear();
            }
            //saying JVM to perform garbage collection
            System.gc();
            return resultBdd;
        }
        if (BDDFactory.and.equals(bddop)) {
            //perform the AND operation
            CustomBDD resultBdd = performAnd(customBDD);
            //reduce the head of the result; non-head vertexes have been reduced already
            resultBdd.reduceVar();
            andCount++;
            if (andCount == 100) {//limit of the AND cache
                andCount = 0;
                factory.currentAndCache.clear();
            }
            //saying JVM to perform garbage collection
            System.gc();
            return resultBdd;
        }
        if (BDDFactory.xor.equals(bddop)) {
            //perform the XOR operation
            CustomBDD resultBdd = performXor(customBDD);
            //reduce the head of the result; non-head vertexes have been reduced already
            resultBdd.reduceVar();
            xorCount++;
            if (xorCount == 100) {//limit of the XOR cache
                xorCount = 0;
                factory.currentXorCache.clear();
            }
            //saying JVM to perform garbage collection
            System.gc();
            return resultBdd;
        }
        //we do not support other binary operations over CustomBDD instances
        throw new UnsupportedOperationException();
    }

    private CustomBDD performOr(CustomBDD bdd) {
        String thisBdd = this.toString();
        String thatBdd = bdd.toString();
        CustomBDD cachedBdd = factory.currentOrCache.get(thisBdd + ";" + thatBdd);
        if (cachedBdd != null) {
            //certain luck: we have already performed OR over the same OBDDs
            Map<Integer, CustomBDDItem> newMap = new HashMap<>();
            for (Entry<Integer, CustomBDDItem> entry : cachedBdd.bddMap.entrySet()) {
                newMap.put(entry.getKey(), new CustomBDDItem(entry.getValue()));
            }
            return new CustomBDD(newMap, factory);
        }
        cachedBdd = factory.currentOrCache.get(thatBdd + ";" + thisBdd);
        if (cachedBdd != null) {
            //certain luck: we have already performed OR over the same OBDDs
            Map<Integer, CustomBDDItem> newMap = new HashMap<>();
            for (Entry<Integer, CustomBDDItem> entry : cachedBdd.bddMap.entrySet()) {
                newMap.put(entry.getKey(), new CustomBDDItem(entry.getValue()));
            }
            return new CustomBDD(newMap, factory);
```

```
}
if (bdd.isOne() || this.isOne()) {
    //constant one gives constant one with any boolean function
    return new CustomBDD(factory.one);
}
if (bdd.isZero()) {
    //constant zero gives the second argument with any boolean function
    return new CustomBDD(this);
}
if (this.isZero()) {
    //constant zero gives the second argument with any boolean function
    return new CustomBDD(bdd);
}
Map<Integer, Integer> variablesOrder = factory.getVariablesOrderMapping();
//getting head variables indexes
Integer thisVar = this.var();
Integer thatVar = bdd.var();
Integer thisVarOrder = variablesOrder.get(thisVar);
Integer thatVarOrder = variablesOrder.get(thatVar);
if (thisVarOrder.equals(thatVarOrder)) {//the head variables are the same
    CustomBDD thisZeroSubstitution = this.replaceVariableWithValue(thisVar, false);
    CustomBDD thisOneSubstitution = this.replaceVariableWithValue(thisVar, true);
    CustomBDD thatZeroSubstitution = bdd.replaceVariableWithValue(thatVar, false);
    CustomBDD thatOneSubstitution = bdd.replaceVariableWithValue(thatVar, true);
    //go one level lower
    CustomBDD stepResultZero = thisZeroSubstitution.performOr(thatZeroSubstitution);
    CustomBDD stepResultOne = thisOneSubstitution.performOr(thatOneSubstitution);
    //merge partly-getted OBDDs
    cachedBdd = ifThenElse(stepResultOne, stepResultZero);
    //put the result into the cache
    factory.currentOrCache.put(thisBdd + ";" + thatBdd, cachedBdd);
    //construct the result OBDD
    Map<Integer, CustomBDDItem> newMap = new HashMap<>();
    for (Entry<Integer, CustomBDDItem> entry : cachedBdd.bddMap.entrySet()) {
        newMap.put(entry.getKey(), new CustomBDDItem(entry.getValue()));
    }
    return new CustomBDD(newMap, factory);
}

if (thisVarOrder < thatVarOrder) {//the head variable of this CustomBDD is smaller
    CustomBDD thisZeroSubstitution = this.replaceVariableWithValue(thisVar, false);
    CustomBDD thisOneSubstitution = this.replaceVariableWithValue(thisVar, true);
    //go one level lower
    CustomBDD stepResultZero = thisZeroSubstitution.performOr(new CustomBDD(bdd));
    CustomBDD stepResultOne = thisOneSubstitution.performOr(bdd);
    //merge partly-getted OBDDs
    cachedBdd = ifThenElse(stepResultOne, stepResultZero);
    //put the result into the cache
    factory.currentOrCache.put(thisBdd + ";" + thatBdd, cachedBdd);
    //construct the result OBDD
    Map<Integer, CustomBDDItem> newMap = new HashMap<>();
    for (Entry<Integer, CustomBDDItem> entry : cachedBdd.bddMap.entrySet()) {
        newMap.put(entry.getKey(), new CustomBDDItem(entry.getValue()));
    }
    return new CustomBDD(newMap, factory);
}

if (thatVarOrder < thisVarOrder) {//the head variable of this CustomBDD is bigger
    CustomBDD thatZeroSubstitution = bdd.replaceVariableWithValue(thatVar, false);
    CustomBDD thatOneSubstitution = bdd.replaceVariableWithValue(thatVar, true);
    //go one level lower
    CustomBDD stepResultZero = performOr(thatZeroSubstitution);
    CustomBDD stepResultOne = performOr(thatOneSubstitution);
    //merge partly-getted OBDDs
    cachedBdd = bdd.ifThenElse(stepResultOne, stepResultZero);
    //put the result into the cache
    factory.currentOrCache.put(thisBdd + ";" + thatBdd, cachedBdd);
    //construct the result OBDD
    Map<Integer, CustomBDDItem> newMap = new HashMap<>();
```

```
            for (Entry<Integer, CustomBDDItem> entry : cachedBdd.bddMap.entrySet()) {
                newMap.put(entry.getKey(), new CustomBDDItem(entry.getValue()));
            }
            return new CustomBDD(newMap, factory);
        }
        return new CustomBDD(factory);
    }

    private boolean reduced = false;

    private void reduceVar() {
        if (!reduced) {
            //we reduce the OBDD only once, because it's partly-immutable
            Integer headVar = var();
            if (headVar >= 0) {//the OBDD is not a constant
                CustomBDDItem headItem = usedVariables.get(headVar).get(0);
                CustomBDDItem childItem = bddMap.get(headItem.getZeroLink());
                Integer headItemIndex = childItem.getParentLink();
                if (headItemIndex == null) {
                    childItem = bddMap.get(headItem.getOneLink());
                    headItemIndex = childItem.getParentLink();
                }
                if (headItemIndex == null) {
                    for (Integer parent : childItem.getAdditionalParents()) {
                        CustomBDDItem nextItem = bddMap.get(parent);
                        if (nextItem == headItem) {
                            headItemIndex = parent;
                            break;
                        }
                    }
                }

                //creating two children OBDDs of the head vertex
                Map<Integer, CustomBDDItem> zeroChildBddMap = new HashMap<>(bddMap);
                Map<Integer, CustomBDDItem> oneChildBddMap = new HashMap<>(bddMap);
                zeroChildBddMap.put(CustomBDDItem.DEFAULT_ZERO_ITEM_KEY,
                        new CustomBDDItem(bddMap.get(CustomBDDItem.DEFAULT_ZERO_ITEM_KEY)));
                zeroChildBddMap.put(CustomBDDItem.DEFAULT_ONE_ITEM_KEY,
                        new CustomBDDItem(bddMap.get(CustomBDDItem.DEFAULT_ONE_ITEM_KEY)));
                //removing the old head
                zeroChildBddMap.remove(headItemIndex);
                deleteAllChildren(headItem.getZeroLink(), headItemIndex, zeroChildBddMap);
                CustomBDDItem oneItem = new
CustomBDDItem(zeroChildBddMap.get(headItem.getOneLink()));
                zeroChildBddMap.put(headItem.getOneLink(), oneItem);
                oneItem.setParentLink(null);
                oneItem.removeAdditionalParent(headItemIndex);

                oneChildBddMap.put(CustomBDDItem.DEFAULT_ZERO_ITEM_KEY,
                        new CustomBDDItem(bddMap.get(CustomBDDItem.DEFAULT_ZERO_ITEM_KEY)));
                oneChildBddMap.put(CustomBDDItem.DEFAULT_ONE_ITEM_KEY,
                        new CustomBDDItem(bddMap.get(CustomBDDItem.DEFAULT_ONE_ITEM_KEY)));
                //removing the old head
                oneChildBddMap.remove(headItemIndex);
                deleteAllChildren(headItem.getOneLink(), headItemIndex, oneChildBddMap);
                CustomBDDItem zeroItem = new
CustomBDDItem(oneChildBddMap.get(headItem.getZeroLink()));
                oneChildBddMap.put(headItem.getZeroLink(), zeroItem);
                zeroItem.setParentLink(null);
                zeroItem.removeAdditionalParent(headItemIndex);

                CustomBDD zeroChildBdd = new CustomBDD(zeroChildBddMap, factory);
                CustomBDD oneChildBdd = new CustomBDD(oneChildBddMap, factory);
                if (zeroChildBdd.equals(oneChildBdd)) {
                    //two immediate child-OBDDs are equal; so, the OBDD does not
                    //depends on the head variable
                    setBddMap(zeroChildBddMap, false);
                }
            }//we do not to reduce the constant OBDD
```

```
                reduced = true;
            }
        }

        /**
         *
         * @param variable variable to replace with given value
         * @param value boolean value to replace given variable with
         * @return the result of replacing
         */
        public CustomBDD replaceVariableWithValue(int variable, boolean value) {
            Map<Integer, CustomBDDItem> newBddMap = new HashMap<>(this.bddMap);
            List<CustomBDDItem> items = this.usedVariables.get(variable);
            if (items == null || items.isEmpty()) {//the OBDD is a constant
                return new CustomBDD(this);
            }
            for (CustomBDDItem item : items) {
                //replacing all items having the given variable with the given value
                Integer necessaryLink = value ? item.getOneLink() : item.getZeroLink();
                Integer unnecessaryLink = value ? item.getZeroLink() : item.getOneLink();
                CustomBDDItem parentItem = newBddMap.get(item.getParentLink());
                CustomBDDItem necessaryChildItem = newBddMap.get(necessaryLink);

                Integer itemLink = necessaryChildItem.getParentLink();
                if (itemLink == null) {
                    List<Integer> terminalParents = necessaryChildItem.getAdditionalParents();
                    if (terminalParents != null && !terminalParents.isEmpty()) {
                        for (Integer terminalParent : terminalParents) {
                            if (newBddMap.get(terminalParent) == item) {
                                itemLink = terminalParent;
                                break;
                            }
                        }
                    }
                }
                newBddMap.remove(itemLink);

                //deleting all unnecessary vertexes
                deleteAllChildren(unnecessaryLink, itemLink, newBddMap);
                CustomBDDItem newNecessaryChildItem = new
CustomBDDItem(newBddMap.get(necessaryLink));
                newNecessaryChildItem.setParentLink(item.getParentLink());
                List<Integer> childAdditionalParents = newNecessaryChildItem.getAdditionalParents();
                if (childAdditionalParents != null) {
                    newNecessaryChildItem.setParentLink(null);
                    newNecessaryChildItem.setAdditionalParents(childAdditionalParents);
                    if (parentItem != null) {
                        newNecessaryChildItem.replaceAdditionalParent(
                                itemLink, item.getParentLink());
                    }
                    else {
                        newNecessaryChildItem.removeAdditionalParent(itemLink);
                    }
                }
                newBddMap.put(necessaryLink, newNecessaryChildItem);

                if (parentItem != null) {
                    CustomBDDItem newParentItem = new CustomBDDItem(
                            parentItem.getZeroLink().equals(itemLink) ? necessaryLink :
parentItem.getZeroLink(),
                            parentItem.getOneLink().equals(itemLink) ? necessaryLink :
parentItem.getOneLink(),
                            parentItem.getVariable(),
                            parentItem.getParentLink());
                    newBddMap.put(item.getParentLink(), newParentItem);
                }
            }

            if (!newBddMap.containsKey(CustomBDDItem.DEFAULT_ZERO_ITEM_KEY)) {
```

```
            //then it is just a constant OBDD
            CustomBDDItem item = newBddMap.get(CustomBDDItem.DEFAULT_ONE_ITEM_KEY);
            if (CustomBDDItem.DEFAULT_ZERO_ITEM_KEY.equals(item.getZeroLink())) {
                return (CustomBDD) factory.zero();
            }
            else {
                return (CustomBDD) factory.one();
            }
        }
        if (!newBddMap.containsKey(CustomBDDItem.DEFAULT_ONE_ITEM_KEY)) {
            //then it is just a constant OBDD
            CustomBDDItem item = newBddMap.get(CustomBDDItem.DEFAULT_ZERO_ITEM_KEY);
            if (CustomBDDItem.DEFAULT_ONE_ITEM_KEY.equals(item.getZeroLink())) {
                return (CustomBDD) factory.one();
            }
            else {
                return (CustomBDD) factory.zero();
            }
        }
        //constructing and returning new OBDD
        return new CustomBDD(newBddMap, this.factory);
    }

}
```

Приложение 2. Часть исходного кода класса KripkeStructureTranslator.

```
public class KripkeStructureTranslator {

    /**
     *
     * @return True if translation was successful (so you can get its results
     * via getters); false otherwise (then see logs for details).
     */
    public boolean tryTranslate() {
        //build states table of the given model
        tableBuilder.buildTable();
        table = tableBuilder.getStatesTable();
        if (table == null) {
            isTranslated = false;
            return isTranslated;
        }
        //get the array of table indexes and sort it
        statesNumbers = table.keySet().toArray(new Integer[0]);
        Arrays.sort(statesNumbers);

        //translate the program then
        isTranslated = translateProgram();
        return isTranslated;
    }

     private boolean translateProgram() {
        //count the number of boolean variables in the OBDD
        setVariablesNumberBDD();
        setFactory();
        //set some utilities objects
        setConditionParser();
        setExpressionParser();
        setCalculator();
        if (!prepareAtomicPredicates()) {
            //error during parsing of atomic predicates
            return false;
        }
        //start the translation; thisPC is the counter over the built table
        int thisPC = statesNumbers[0];
        statesNumbers = null;
        List<Object> current = new ArrayList<>();
        for (int i = 0; i < variablesNumber; i++) {
            current.add(null);
```

```
        }
        setStartState(thisPC);
        statesTransitionBDD = factory.zero();
        StateType stateType;
        try {
            stateType = getNextStateType(thisPC);
        }
        catch (IllegalArgumentException ex) {
            logger.log(Level.INFO, "Cancelling translation");
            return false;
        }
        switch (stateType) {
            case ASSIGN:
                return translateAssign(current, thisPC);

            case ELSE:
                logger.log(Level.ERROR, "Else without if");
                logger.log(Level.INFO, "Cancelling translation");
                return false;

            case END_OF_GROUP:
                logger.log(Level.ERROR, "Close bracket without context");
                logger.log(Level.INFO, "Cancelling translation");
                return false;

            case END_OF_PROGRAM:
                logger.log(Level.ERROR, "Empty program");
                logger.log(Level.INFO, "Cancelling translation");
                return false;

            case IF:
                logger.log(Level.INFO, "Transalting if");
                return translateIf(current, thisPC);

            case READ:
                logger.log(Level.INFO, "Translating read");
                return translateRead(current, thisPC);

            case WHILE:
                logger.log(Level.INFO, "Translating while");
                return translateWhile(current, thisPC);
        }
        return false;
    }

    private boolean translateIf(List<Object> current, int pc) {
        //get OBDD describing the current state
        currentStateBDD = translateState(current, pc);
        //get OBDD decribing the transition to the next state
        currentTransitionBDD = translateState(current, pc);
        List<Object> transition = table.get(pc);
        //get necessary information to go futher
        String comment = (String)transition.get(doubleVariablesNumber + 2);
        int semicolonPosition = comment.lastIndexOf(SEMICOLON);
        String condition = comment.substring(IF_SHIFT, semicolonPosition);
        int thenPC = (Integer)transition.get(doubleVariablesNumber + 1);
        int elsePC = Integer.parseInt(comment.substring(semicolonPosition + ELSE_SHIFT + 1));
        Map<Variable, String> variableValues = new HashMap<>();
        for (int i = 0; i < variables.size(); i++) {
            if (current.get(i) == null) {
                continue;
            }
            variableValues.put(variables.get(i), current.get(i).toString());
        }
        String conditionValueString = calculator.calculate(null, condition,
                conditionParser.parseCondition(condition));
        if (conditionValueString == null) {
            logger.log(Level.ERROR, "Bad condition " + condition);
            return false;
```

```
}
//check the condition in the IF in the given model
boolean conditionValue;
if (conditionValueString.equals(TRUE)) {
    conditionValue = true;
}
else {
    if (conditionValueString.equals(FALSE)) {
        conditionValue = false;
    }
    else {
        logger.log(Level.ERROR, "Bad condition " + condition);
        return false;
    }
}
if (conditionValue) {//translate THEN branch
    currentTransitionBDD.andWith(translateNextState(current, thenPC));
    addToStatesTransitionBDD();
    addToPredicatesBDD(variableValues);
    variableValues.clear();
    variableValues = null;
    StateType stateType;
    try {
        stateType = getNextStateType(thenPC);
    }
    catch (IllegalArgumentException ex) {
        logger.log(Level.INFO, "Cancelling translation");
        return false;
    }
    switch (stateType) {//the type of the first state at the THEN block
        case ASSIGN:
            logger.log(Level.INFO, "Translating assign");
            return translateAssign(current, thenPC);

        case ELSE:
            logger.log(Level.INFO, "Translating else");
            return translateElse(current, thenPC);

        case END_OF_GROUP:
            logger.log(Level.INFO, "Translating end of group");
            return translateEndOfGroup(current, thenPC);

        case END_OF_PROGRAM:
            logger.log(Level.INFO, "Translating end of program");
            return translateEndOfProgram(current, thenPC);

        case IF:
            logger.log(Level.INFO, "Transalting if");
            return translateIf(current, thenPC);

        case READ:
            logger.log(Level.INFO, "Translating read");
            return translateRead(current, thenPC);

        case WHILE:
            logger.log(Level.INFO, "Translating while");
            return translateWhile(current, thenPC);
    }
}
else {//translate ELSE branch; or smth goes after the end of THEN
    currentTransitionBDD.andWith(translateNextState(current, elsePC));
    addToStatesTransitionBDD();
    addToPredicatesBDD(variableValues);
    variableValues.clear();
    variableValues = null;
    StateType stateType;
    try {
        stateType = getNextStateType(elsePC);
    }
```

```
            catch (IllegalArgumentException ex) {
                logger.log(Level.INFO, "Cancelling translation");
                return false;
            }
            switch (stateType) {//the type of the first state at the ELSE block
                case ASSIGN:
                    logger.log(Level.INFO, "Translating assign");
                    return translateAssign(current, elsePC);

                case ELSE:
                    logger.log(Level.INFO, "Translating else");
                    return translateElse(current, elsePC);

                case END_OF_GROUP:
                    logger.log(Level.INFO, "Translating end of group");
                    return translateEndOfGroup(current, elsePC);

                case END_OF_PROGRAM:
                    logger.log(Level.INFO, "Translating end of program");
                    return translateEndOfProgram(current, elsePC);

                case IF:
                    logger.log(Level.INFO, "Transalting if");
                    return translateIf(current, elsePC);

                case READ:
                    logger.log(Level.INFO, "Translating read");
                    return translateRead(current, elsePC);

                case WHILE:
                    logger.log(Level.INFO, "Translating while");
                    return translateWhile(current, elsePC);
            }
        }//there was an error, unknown successor
        return false;
    }


}
```

Приложение 3. Скрипт создания БД приложения.

```
create extension citext;

create extension plperl;
create language plperlu;

create or replace function check_email(email text) returns bool
language plperlu
as $$
use Email::Address;

my @addresses = Email::Address->parse($_[0]);
return scalar(@addresses) > 0 ? 1 : 0;
$$;

create table systemclients (
id serial primary key,
busy boolean not null default false,
email citext unique not null
check (check_email(email)));

create table project (
id serial primary key,
user_id int not null references systemclients(id),
title text not null,
active boolean not null default true
);

create table result (
```

```
id serial primary key,
result_name varchar(100) not null unique
);

create table projecthistory (
id serial primary key,
project_id int not null references project(id),
listing text not null,
property text not null,
counter_examples text,
result_id int not null references result(id)
);

insert into result(result_name) (select 'Property holds' union
                  select 'Property does not hold' union
                  select 'Error: input is not correct' union
                  select 'Error during the verification');
```

Приложение 4. Асинхронный вызов верификации на сервере.

Фрагмент класса ProjectFacade.

```
@Stateless
@Path("/project")
public class ProjectFacade extends AbstractFacade<Project> {

    @EJB
    private SystemClientFacade clientFacade;

    @EJB
    private ProjectHistoryFacade projectHistoryFacade;

    @EJB
    private VerificationResultFacade resultFacade;

    @POST
    @Path("/check")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Response runModelChecking(String json) {
        Record record = parseMessage(json);
        if (record == null) {
            return Response.ok(createMessage(null), MediaType.APPLICATION_JSON).build();
        }
        ProjectDescriptor projectDescriptor = record.getProjectDescriptor();
        if (projectDescriptor == null || projectDescriptor.getId() == null) {
            return Response.ok(createMessage(null), MediaType.APPLICATION_JSON).build();
        }
        Long projectId = projectDescriptor.getId();
        Project project = findById(projectId);
        if (project == null || !project.getActive()) {
            return Response.ok(createMessage(null), MediaType.APPLICATION_JSON).build();
        }
        if (project.getSystemClient() == null || project.getSystemClient().getBusy()) {
            return Response.ok(createMessage(null), MediaType.APPLICATION_JSON).build();
        }
        project.getSystemClient().setBusy(Boolean.TRUE);
        clientFacade.edit(project.getSystemClient());
        resultFacade.runModelChecking(projectDescriptor, project);
        return Response.ok(createMessage(record), MediaType.APPLICATION_JSON).build();
    }

}
```

Фрагмент класса VerificationResultFacade.

```
@Stateless
```

```
public class VerificationResultFacade extends AbstractFacade<Result> {

    private final MailSender mailSender = new SmtpMailSender();

    @EJB
    private ProjectHistoryFacade projectHistoryFacade;

    @EJB
    private SystemClientFacade clientFacade;

    @EJB
    private ProjectFacade projectFacade;
    @Asynchronous
    public void runModelChecking(ProjectDescriptor descriptor, Project project) {
        Controller systemController = new Controller();
        if (descriptor == null || descriptor.getProgram() == null
                || descriptor.getProperty() == null
                || descriptor.getProperty().isEmpty())
        {
            List<Result> result = entityManager.createNamedQuery("Result.findByName")
                    .setParameter(Result.NAME_QUERY_PARAMETER, VerificationResult.WRONG_INPUT)
                    .getResultList();
            Result resultOfVerification = result.get(DEFAULT_RESULT_INDEX);
            notifyUser(project.getSystemClient().getEmail(),
                    descriptor == null || descriptor.getProgram() == null ? "" : de-
scriptor.getProgram(),
                    descriptor == null || descriptor.getProperty() == null ? "" : de-
scriptor.getProperty(),
                    resultOfVerification.getResultName(), null);
            ProjectHistory newAttempt = new ProjectHistory();
            newAttempt.setProperty(descriptor == null || descriptor.getProperty() == null
                    ? "" : descriptor.getProperty());
            newAttempt.setListing(descriptor == null || descriptor.getProgram() == null
                    ? "" : descriptor.getProgram());
            newAttempt.setResult(resultOfVerification);
            newAttempt.setProject(project);
            newAttempt.setCounterExamples(null);
            projectHistoryFacade.create(newAttempt);
            project.getSystemClient().setBusy(Boolean.FALSE);
            clientFacade.edit(project.getSystemClient());
            projectFacade.edit(project);
            return;
        }
        List<String> properties = descriptor.getAtomicPredicates();
        systemController.setAtomicPredicates(properties);
        systemController.setCTLFormula(descriptor.getProperty());
        systemController.setProgram(descriptor.getProgram());
        CTLVerificator verificator = systemController.startVerification();
        VerificationResult verificationResult;
        String result;
        if (verificator == null) {
            verificationResult = null;
            result = VerificationResult.WRONG_INPUT;
        }
        else {
            verificationResult = verificator.getVerificationResult();
            result = verificationResult.toDatabaseString();
        }

        List<Result> resultList = entityManager.createNamedQuery("Result.findByName")
                    .setParameter(Result.NAME_QUERY_PARAMETER, result)
                    .getResultList();
        List<String> counterExamplesList = null;
        if (verificator != null) {
            counterExamplesList =
verificator.getCounterexamples(DEFAULT_NUMBER_OF_COUNTER_EXAMPLES);
        }
        String counterExamples = listToString(counterExamplesList);
        String email = project.getSystemClient().getEmail();
```

```
        String dbProperty = createProperty(descriptor.getProperty(),
                descriptor.getAtomicPredicates());
        notifyUser(email, descriptor.getProgram(), dbProperty,
                result, counterExamples);
        Result resultOfVerification = resultList.get(DEFAULT_RESULT_INDEX);
        ProjectHistory newAttempt = new ProjectHistory();
        newAttempt.setProperty(dbProperty);
        newAttempt.setListing(descriptor.getProgram());
        newAttempt.setResult(resultOfVerification);
        newAttempt.setProject(project);
        newAttempt.setCounterExamples(counterExamples);
        projectHistoryFacade.create(newAttempt);
        project.getSystemClient().setBusy(Boolean.FALSE);
        clientFacade.edit(project.getSystemClient());
        projectFacade.edit(project);
    }

    private void notifyUser(String email, String program, String property,
            String result, String counterExamples)
    {
        String text = EMAIL_MODEL_BODY_TEMPLATE + program + "\n";
        text += EMAIL_PROPERTY_BODY_TEMPLATE + property + "\n";
        text += EMAIL_RESULT_BODY_TEMPLATE + result;
        if (counterExamples != null && !counterExamples.isEmpty()) {
            text += "\n" + EMAIL_COUNTER_EXAMPLES_BODY_TEMPLATE + counterExamples;
        }
        mailSender.sendEmail(email, EMAIL_SUBJECT, text);
    }

}
```

Приложение 5. Часть исходного кода GWT-модуля.

Фрагмент класса DataGridTable (реализация наследника CellTable с возможностью динамического добавления и удаления рядов, а также изменения ячеек).

```
public class DataGridTable<T> extends CellTable<T> {

    public static interface GetValue<T, C> {
        C getValue(T row);
    }

    public static interface ChangeHandler<T, C> {
        void commitChange(T row, C newValue);
    }

    public static interface ButtonClickedHandler<T> {
        void handle(T row);
    }

    protected final List<AbstractEditableCell<?, ?>> editableCells =
            new ArrayList<>();

    private final ListDataProvider<T> dataProvider;

    public DataGridTable() {
        dataProvider = new ListDataProvider<>();
        dataProvider.addDataDisplay(this);
    }

    public <C> Column<T, C> addColumn(Cell<C> cell, String headerText,
      final GetValue<T, C> getter, FieldUpdater<T, C> fieldUpdater) {
        Column<T, C> column = new Column<T, C>(cell) {
            @Override
            public C getValue(T object) {
                return getter.getValue(object);
            }
        };
```

```
        column.setFieldUpdater(fieldUpdater);
        if (cell instanceof AbstractEditableCell<?, ?>) {
            editableCells.add((AbstractEditableCell<?, ?>) cell);
        }
        addColumn(column, headerText);
        return column;
    }

    public void addRow(T row) {
        dataProvider.getList().add(row);
        dataProvider.flush();
    }

    public void removeRow(T row) {
        dataProvider.getList().remove(row);
        dataProvider.flush();
    }

    public void setData(List<T> data) {
        dataProvider.setList(data);
        dataProvider.flush();
    }

}
```

Фрагмент класса Requester, отвечающего за общение с серверной частью приложения.

```
public abstract class Requester {

    private Record answer;
    private ResponseReceivedHandler handler;
    private static final String MEDIA_CONTENT_KEY = "Content-Type";
    private static final String MEDIA_CONTENT_VALUE = "application/json";

    public enum RequestMethod {
        GET, PUT, POST,
        DELETE, HEAD
    }

    public void setResponseReceivedHandler(ResponseReceivedHandler handler) {
        this.handler = handler;
    }

    public void sendRequest(Record record) {
        RequestMethod requestMethod = getRequestMethod();
        Method method = null;
        switch (requestMethod) {
            case DELETE:
                method = RequestBuilder.DELETE;
                break;

            case GET:
                method = RequestBuilder.GET;
                break;

            case HEAD:
                method = RequestBuilder.HEAD;
                break;

            case POST:
                method = RequestBuilder.POST;
                break;

            case PUT:
                method = RequestBuilder.PUT;
                break;
        }
```

```
    if (method == null) {
        if (handler != null) {
            handler.onSuccess(null);
            return;
        }
        else {
            return;
        }
    }
    RequestBuilder requestBuilder = new RequestBuilder(method, getPath());
    if (method == RequestBuilder.POST || method == RequestBuilder.PUT) {
        requestBuilder.setHeader(MEDIA_CONTENT_KEY, MEDIA_CONTENT_VALUE);
        requestBuilder.setRequestData(getRecordAsJson(record));
    }
    requestBuilder.setCallback(new RequestCallback() {

        @Override
        public void onResponseReceived(Request request, Response response) {
            String responseBody = response.getText();
            answer = getJsonAsRecord(responseBody);
            if (handler != null) {
                handler.onSuccess(answer);
            }
        }

        @Override
        public void onError(Request request, Throwable exception) {
            answer = null;
            if (handler != null) {
                handler.onFailure(answer);
            }
        }
    });
    try {
        requestBuilder.send();
    }
    catch (RequestException ex) {
        answer = null;
        if (handler != null) {
            handler.onFailure(answer);
        }
    }
}

public abstract String getPath();
public abstract RequestMethod getRequestMethod();

public void setQueryParametersString(String queryParameters) {

}

private String getRecordAsJson(Record record) {
    String jsonValue = RecordEncoderDecoder.encode(record);
    String json = "{\"" + Record.RECORD_KEY + "\":\"" + jsonValue + "\"}";
    return json;
}

private Record getJsonAsRecord(String json) {
    JSONValue value = JSONParser.parseStrict(json);
    if (value == null) {
        return null;
    }
    JSONObject object = value.isObject();
    if (object == null) {
        return null;
    }
    value = object.get(Record.RECORD_KEY);
    if (value == null) {
        return null;
```

```
        }
        JSONString string = value.isString();
        if (string == null) {
            return null;
        }
        String payload = string.stringValue();
        return RecordEncoderDecoder.decode(payload);
    }

}
```

Приложение 6. WelcomeActivity клиента под Android.

Класс WelcomeActivity.

```
public class WelcomeActivity extends AppCompatActivity implements View.OnClickListener {

    private static final String WRONG_EMAIL_TITLE = "Warning";
    private static final String WRONG_EMAIL_MESSAGE = "The email is not valid";
    private static final String OK_BUTTON_TEXT = "Ok";

    private Button enterButton;
    private EditText emailEditTextField;

    private String email;
    private boolean resumed = false;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_welcome);

        enterButton =(Button)findViewById(R.id.enterButton);
        enterButton.setOnClickListener(this);

        emailEditTextField = (EditText) findViewById(R.id.emailEditText);
    }

    @Override
    public void onClick(View v) {
        switch(v.getId()){
            case R.id.enterButton:
                email = emailEditTextField.getText().toString();
                if (Validator.getInstance().isEmailValid(email)) {
                    startProjectsActivity();
                }
                else {
                    final AlertDialog.Builder builder = new AlertDialog.Builder(this);
                    builder.setTitle(WRONG_EMAIL_TITLE);
                    builder.setMessage(WRONG_EMAIL_MESSAGE);

                    // Set up the buttons
                    builder.setNeutralButton(OK_BUTTON_TEXT, new
DialogInterface.OnClickListener() {
                        @Override
                        public void onClick(DialogInterface dialog, int which) {
                            dialog.cancel();
                        }
                    });
                    builder.show();
                }

        }
    }

    private void startProjectsActivity() {
        Record record = new Record();
        ClientDescriptor clientDescriptor = new ClientDescriptor();
```

```
        clientDescriptor.setClientEmail(email);
        record.setClientDescriptor(clientDescriptor);
        Orchestrator.getInstance().go(record, Orchestrator.DesiredActionType.VIEW_PROJECTS);
        Intent projectsIntent = new Intent(this, ProjectsActivity.class);
        startActivity(projectsIntent);
    }

    @Override
    protected void onResume() {
        super.onResume();
        if (resumed) {
            emailEditTextField.setText("");
        }
        else {
            resumed = true;
        }
    }
}
```

Xml-файл activity_welcome.

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.arcquim.system.android.WelcomeActivity"
    android:background="@color/windowBackground">

    <TextView
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:text="@string/welcomePageTitle"
        android:layout_centerHorizontal="true"
        android:gravity="center"
        android:textSize="@dimen/model_checking_system_text_size"
        android:textAlignment="center"
        android:layout_marginTop="@dimen/margin_top"
        android:textColor="@color/textColor"></TextView>

    <TextView
        android:id="@+id/yourEmailTextView"
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:text="@string/yourEmail"
        android:gravity="center"
        android:layout_centerVertical="true"
        android:textAlignment="center"
        android:textColor="@color/textColor"
        android:textSize="@dimen/text_size"></TextView>

    <EditText
        android:id="@+id/emailEditText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:singleLine="true"
        android:layout_below="@+id/yourEmailTextView"
        android:layout_marginRight="@dimen/edit_text_margin_left_right"
        android:layout_marginLeft="@dimen/edit_text_margin_left_right"
        android:hint="@string/emailHint"
        android:textColor="@color/textColor"
        android:layout_marginTop="@dimen/little_margin"></EditText>

    <Button
        android:id="@+id/enterButton"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/buttonEnterText"
        android:layout_below="@+id/emailEditText"
        android:layout_centerHorizontal="true"
```

```
        android:background="@color/colorAccent"
        android:textColor="@color/textColor"
        android:layout_marginTop="@dimen/little_margin"
        android:layout_marginLeft="@dimen/edit_text_margin_left_right"
        android:layout_marginRight="@dimen/edit_text_margin_left_right"/>
</RelativeLayout>
```

Приложение 7. Unit-тесты некоторых классов.

Класс CustomBDD (начатая реализация библиотеки для работы с OBDD).

```
public class CustomBDDTest {

    @Test
    public void testIsZero() {
        CustomFactory factory = new CustomFactory();
        factory.setVarNum(4);
        BDD zeroBDD = factory.zero();
        assertTrue(zeroBDD.isZero());
    }

    @Test
    public void testIsOne() {
        CustomFactory factory = new CustomFactory();
        factory.setVarNum(4);
        BDD oneBdd = factory.one();
        assertTrue(oneBdd.isOne());
    }

    @Test
    public void testVar() {
        CustomFactory factory = new CustomFactory();
        factory.setVarNum(5);
        BDD bdd = factory.one;
        assertEquals(bdd.var(), -1);
        bdd = factory.nithVar(1);
        assertEquals(bdd.var(), 1);
        bdd = bdd.or(factory.ithVar(0));
        assertEquals(bdd.var(), 0);
    }

    @Test
    public void testNot() {
        CustomFactory factory = new CustomFactory();
        factory.setVarNum(4);
        BDD oneBdd = factory.one();
        assertTrue(oneBdd.not().isZero());
    }

    @Test
    public void testExist() {
        CustomFactory factory = new CustomFactory();
        factory.setVarNum(4);
        BDD bdd = factory.ithVar(0).or(factory.nithVar(1)).and(factory.ithVar(3));
        BDD bddForExist = factory.ithVar(3);
        BDD actualResult = bdd.exist(bddForExist);
        BDD expectedResult = factory.nithVar(0).and(factory.nithVar(1))
                .or(factory.ithVar(0));
        //<0:0, 1:0><0:1>
        assertEquals(expectedResult, actualResult);
    }

    @Test
    public void testForAll() {
        CustomFactory factory = new CustomFactory();
        factory.setVarNum(4);
        BDD bdd = factory.ithVar(0).or(factory.nithVar(1)).and(factory.ithVar(3));
        BDD bddForAll = factory.ithVar(3);
```

```
        BDD actualResult = bdd.forAll(bddForAll);
        BDD expectedResult = factory.zero();
        assertEquals(expectedResult, actualResult);
    }

    @Test
    public void testOr() {
        CustomFactory factory = new CustomFactory();
        factory.setVarNum(6);
        BDD firstBdd = factory.nithVar(2).xor(factory.ithVar(0)).and(factory.ithVar(5)).not();
        BDD secondBdd = factory.ithVar(4).and(factory.ithVar(2)).or(factory.nithVar(0));
        BDD actualResult = firstBdd.or(secondBdd);
        BDD actualReverseResult = secondBdd.or(firstBdd);
        BDD expectedResult = factory.nithVar(0).or(factory.ithVar(0).and(factory.nithVar(2)))

.or(factory.ithVar(0).and(factory.ithVar(2)).and(factory.nithVar(4)).and(factory.nithVar(5)))
                .or(factory.ithVar(0).and(factory.ithVar(2)).and(factory.ithVar(4)));
        //<0:0><0:1, 2:0><0:1, 2:1, 4:0, 5:0><0:1, 2:1, 4:1>;
        assertEquals(expectedResult, actualResult);
        assertEquals(expectedResult, actualReverseResult);
        assertEquals(actualReverseResult, actualResult);

        firstBdd = factory.one();
        secondBdd = factory.zero();
        assertEquals(firstBdd.or(secondBdd), factory.one);

        firstBdd = factory.ithVar(2);
        BDD newResult = actualResult.or(firstBdd);
        expectedResult = factory.one();
        assertEquals(expectedResult, newResult);
    }

}
```

Класс Controller (модуль-верификатор).

```
public class ControllerTest {

    public ControllerTest() {
    }

    /**
     * Test of startVerification method, of class Controller.
     */
    @Test
    public void testStartVerification() {
        Controller instance = new Controller();
        instance.setProgram("int a; int b; read(a); b = a");
        List<String> atomicPredicates = new ArrayList<>();
        atomicPredicates.add("a != b");
        instance.setAtomicPredicates(atomicPredicates);
        instance.setCTLFormula("EF 0");
        CTLVerificator verificator = instance.startVerification();
        VerificationResult result = verificator.getVerificationResult();
        VerificationResult expResult = VerificationResult.PROPERTY_NOT_HOLDS;
        assertEquals(expResult, result);
    }

}
```