Министерство образования и науки Российской Федерации

УДК 004.4’242
ГРНТИ 20.01.01, 28.23.29
Инв. № 310275

УТВЕРЖДЕНО:
Исполнитель: **федеральное государственное бюджетное образовательное учреждение высшего профессионального образования «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»**
Ректор ФГБОУ ВПО «СПбНИУ ИТМО» **_____/ В.Н. Васильев/** **М.П.**

НАУЧНО-ТЕХНИЧЕСКИЙ ОТЧЕТ

о выполнении 2 этапа Государственного контракта
№ 16.740.11.0455 от 13 мая 2011 г.

Исполнитель: федеральное государственное бюджетное образовательное учреждение высшего профессионального образования «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»
Программа (мероприятие): Федеральная целевая программа «Научные и научно-педагогические кадры инновационной России» на 2009-2013 гг., в рамках реализации мероприятия № 1.2.1 Проведение научных исследований научными группами под руководством докторов наук.
Проект: Разработка метода машинного обучения на основе алгоритмов решения задачи о выполнимости булевой формулы для построения управляющих конечных автоматов
Руководитель проекта: Шалыто Анатолий Абрамович

Санкт-Петербург
2011 г.

СПИСОК ОСНОВНЫХ ИСПОЛНИТЕЛЕЙ

по Государственному контракту 16.740.11.0455 от 13 мая 2011 на выполнение поисковых научно-исследовательских работ для государственных нужд

Организация-Исполнитель: федеральное государственное бюджетное образовательное учреждение высшего профессионального образования «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

Руководитель темы:

доктор технических наук, профессор	А.А. Шалыто

Исполнители темы:

без ученой степени, без ученого звания	В.И. Ульянцев
без ученой степени, без ученого звания	Ф. Н. Царев
без ученой степени, без ученого звания	М. В. Буздалов
кандидат технических наук, доцент	Г. А. Корнеев
кандидат технических наук, без ученого звания	Е.Г. Князев
без ученой степени, без ученого звания	С.В. Казаков

РЕФЕРАТ

Отчет 48 с., 5 гл., 11 рис., 4 табл., 15 источников.

Ключевые слова: машинное обучение, управляющий конечный автомат, автоматное программирование, задача о выполнимости булевой формулы, дерево сценариев, граф совместимости, теория сложности, *NP*-полнота.

В настоящем отчете излагаются результаты выполнения *поисковых научно-исследовательских работ по теме «Разработка метода машинного обучения на основе алгоритмов решения задачи о выполнимости булевой формулы для построения управляющих конечных автоматов»*, выполняемых в рамках государственного контракта, заключенного между Министерством образования и науки Российской Федерации и государственным образовательным учреждением высшего профессионального образования «Санкт-Петербургский государственный университет информационных технологий, механики и оптики» в соответствии с решением Конкурсной комиссии Министерства образования и науки Российской Федерации № 1 (протокол от 25.03.2011 № 3/0173100003711000031) по лоту шифр «2011-1.2.1-113-002» «Проведение научных исследований научными группами под руководством докторов наук в области информатики» в рамках федеральной целевой программы «Научные и научно-педагогические кадры инновационной России» на 2009 – 2013 годы, утвержденной постановлением Правительства Российской Федерации от 28 июля 2008 года № 568 «О федеральной целевой программе «Научные и научно-педагогические кадры инновационной России» на 2009–2013 годы».

Целями настоящего этапа являются:

1. Теоретическая оценка сложности задачи построения управляющего автомата по сценариям работы программы.
2. Разработка алгоритма построения дерева сценариев работы.
3. Программная реализация алгоритма построения дерева сценариев работы.
4. Разработка алгоритма построения графа совместимости вершин дерева сценариев.
5. Программная реализация алгоритма построения графа совместимости вершин дерева сценариев.

Излагаются результаты теоретической оценки сложности задачи построения управляющего автомата по сценариям работы, а именно – доказывается *NP*-полнота данной задачи.

Приводится описание и программная реализация алгоритма построения дерева сценариев и алгоритма построения графа совместимости вершин дерева сценариев.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В настоящем отчете излагаются результаты выполнения *поисковых научно-исследовательских работ по теме «Разработка метода машинного обучения на основе алгоритмов решения задачи о выполнимости булевой формулы для построения управляющих конечных автоматов»*, выполняемых в рамках государственного контракта, заключенного между Министерством образования и науки Российской Федерации и государственным образовательным учреждением высшего профессионального образования «Санкт-Петербургский государственный университет информационных технологий, механики и оптики» в соответствии с решением Конкурсной комиссии Министерства образования и науки Российской Федерации № 1 (протокол от 25.03.2011 № 3/0173100003711000031) по лоту шифр «2011-1.2.1-113-002» «Проведение научных исследований научными группами под руководством докторов наук в области информатики» в рамках федеральной целевой программы «Научные и научно-педагогические кадры инновационной России» на 2009 – 2013 годы, утвержденной постановлением Правительства Российской Федерации от 28 июля 2008 года № 568 «О федеральной целевой программе «Научные и научно-педагогические кадры инновационной России» на 2009–2013 годы».

Целями настоящего этапа являются:

1. Теоретическая оценка сложности задачи построения управляющего автомата по сценариям работы программы.
2. Разработка алгоритма построения дерева сценариев работы.
3. Программная реализация алгоритма построения дерева сценариев работы.
4. Разработка алгоритма построения графа совместимости вершин дерева сценариев.
5. Программная реализация алгоритма построения графа совместимости вершин дерева сценариев.

Отчет имеет следующую структуру. В первой главе приводятся результаты выполнения теоретической оценки сложности поставленной задачи. Доказывается принадлежность задачи сложностным классам *NP*, *NP*-трудных задач, *NP*-полных задач.

Во второй главе приводится описание алгоритма построения дерева сценариев работы. В третьей главе приводится программная реализация разработанного алгоритма.

В четвертой главе приводится описание алгоритма построения графа совместимости вершин дерева сценариев. В пятой главе приводится программная реализация разработанного алгоритма.

Каждая из глав снабжена выводами, кратко резюмирующими содержание главы. В заключении дается общая оценка работ по этапу.

Машинное обучение [3] – современный раздел информатики, посвященный созданию алгоритмов, которые обучают параметры некоторой модели на заранее подготовленных тестовых примерах, либо в процессе моделирования ее работы в некоторой внешней среде (обучение «на собственных ошибках»).

Задача о выполнимости булевой формулы – одна из важнейших задач информатики. Она состоит в том, чтобы по заданной булевой формуле найти такой набор значений входящих в нее переменных, что результат вычисления формулы – «истина». Эта задача относится к классу *NP*-полных задач, но для ее решения существует большое число весьма эффективных на практике программных средств. Наиболее современные из этих программных средств такие, как *zChaff* [15], *MiniSat* [11], *CryptoMiniSat* [10], *Plingeling* [8] способны обрабатывать формулы длиной в несколько миллионов символов, содержащие десятки и сотни тысяч переменных.

С использованием этих программных средств в настоящее время решаются такие задачи, как проверка эквивалентности формальных моделей программ, верификации моделей программ, фор-

мальной верификации микропроцессоров, генерации тестовых шаблонов, разводка печатных плат, и т. д.

Автоматное программирование [4] – парадигма программирования, при использовании которой программу предлагается строить в виде совокупности автоматизированных объектов управления, каждый из которых содержит систему управления (взаимодействующие конечные автоматы) и объект управления.

Объект управления характеризуется множеством вычислительных состояний, а также двумя наборами функций: множеством предикатов, отображающих вычислительное состояние в логическое значение (истина или ложь), и множеством действий, позволяющих изменять вычислительное состояние. Управляющий автомат определяется конечным множеством управляющих состояний, функцией переходов и функцией действий.

Исследования, проводимые в ряде университетов мира (например, в Стэнфордском университете), показывают, что модели, основанные на состояниях, целесообразно применять для построения систем со сложным поведением. Такие системы могут в ответ на одно и то же входное воздействие вырабатывать различные выходные воздействия в зависимости от предыстории работы.

Модели, основанные на состояниях, широко применяются в машинном обучении. Примером таких моделей являются Марковские цепи и скрытые Марковские модели [6]. Они могут применяться в таких задачах, как распознавание речи, и для них разработан ряд алгоритмов обучения. Область их применения ограничивается тем, что эти модели имеют вероятностный характер. Наиболее близкими к ним моделями, имеющими детерминированный характер, являются конечные автоматы.

Важным достоинством автоматных программ является возможность их автоматической верификации [1], что является очень существенным свойством для систем с повышенными требованиями к надежности. Для этих систем важную роль играет влияние человеческого фактора на процесс разработки. Поэтому актуальной задачей является разработка методов машинного обучения для построения конечных автоматов.

Как правило, в машинном обучении конечные автоматы рассматриваются как распознаватели регулярных языков [14], или как преобразователи слов одного регулярного языка в слова другого. В случае построения управляющих автоматов обычно рассматриваются системы с малым числом входных переменных (одной или двумя), что существенно ограничивает область применения таких алгоритмов.

Кроме этого, большая часть существующих алгоритмов машинного обучения для конечных автоматов основаны на принципе обучения «на собственных ошибках» и не являются достаточно эффективными для ряда задач, так как не позволяют эффективно учитывать знания человека. Те алгоритмы, которые основаны на принципе обучения на примерах, обладают недостаточно высокой производительностью для использования на практике – они позволяют работать с автоматами, содержащими пять-десять состояний, а суммарная длина обучающих примеров может составлять не более двухсот-трехсот событий. Построение автомата с помощью существующих алгоритмов может занимать от нескольких минут до нескольких дней.

Для устранения недостатков существующих методов будет разработан метод машинного обучения, основанный на решении задачи о выполнимости булевой формулы. В качестве входных данных этот метод будет использовать сценарии работы – один из типов обучающих примеров (тестов). Предварительные исследования показывают, что новый метод будет показывать существенно более высокую производительность (в 10-100 раз более высокую, чем у существующих методов) и сможет обрабатывать автоматы с десятками состояний, а размер входных данных сможет составлять тысячи событий.

Изложенное позволяет утверждать, что результаты выполнения научно-исследовательской работы будут превышать мировой уровень разработок в рассматриваемой области.

1. ТЕОРЕТИЧЕСКАЯ ОЦЕНКА СЛОЖНОСТИ ЗАДАЧИ ПОСТРОЕНИЯ УПРАВЛЯЮЩЕГО АВТОМАТА ПО СЦЕНАРИЯМ РАБОТЫ ПРОГРАММЫ

В настоящем разделе приводится теоретическая оценка сложности построения автомата по сценариям работы программы – доказывается *NP*-полнота поставленной задачи.

1.1. Постановка задачи

Приведем постановку задачи, теоретическую сложность которой будем оценивать.

Управляющим конечным автоматом называется детерминированный конечный автомат, каждый переход которого помечен *событием*, последовательностью *выходных воздействий* и *охранным условием*, представляющим собой логическую формулу от *входных переменных*.

Автомат получает события от так называемых *поставщиков событий* (в их роли могут выступать внешняя среда, интерфейс пользователя и т. д.) и генерирует выходные воздействия для *объекта управления*. При поступлении события автомат выполняет переход в соответствии с охранными условиями и значениями входных переменных. При выполнении перехода генерируются выходные воздействия, которыми он помечен, и автомат переходит в соответствующее состояние. Отметим, что состояния такого автомата не делятся на допускающие и не допускающие. Пример управляющего автомата приведен на рис. 1.

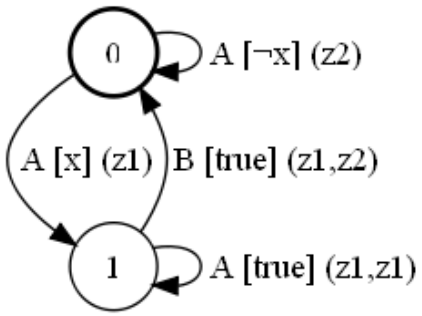

Рис. 1. Пример управляющего конечного автомата

Для данного автомата множество входных событий равно $\{A, B\}$, охранные условия зависят от единственной логической входной переменной x, множество выходных воздействий равно $\{z1, z2\}$. Далее, состояние автомата с номером 0 будем считать начальным.

В качестве исходных данных для построения управляющего конечного автомата используется множество *сценариев работы*. Сценарием работы будем называть последовательность $T_1 \ldots T_n$ троек $T_i = \langle e_i, f_i, A_i \rangle$, где e_i – входное событие, f_i – булева формула от входных переменных, задающая охранное условие, A_i – последовательность выходных воздействий. В дальнейшем тройки T_i будем называть *элементами сценария*.

Будем говорить, что автомат, находясь в состоянии *state*, *удовлетворяет элементу сценария* T_i, если из *state* исходит переход, помеченный событием e_i, последовательностью выходных воздействий A_i и охранным условием, тождественно равным f_i как булева формула. Автомат *удовлетворяет сценарию работы* $T_1 \ldots T_n$, если он удовлетворяет каждому элементу данного сценария, находясь при этом в состояниях пути, образованного соответствующими переходами.

Решается задача построения управляющего конечного автомата с заданным числом состояний C по заданному множеству сценариев работы Sc, которым автомат должен удовлетворять.

1.2. Доказательство *NP*-полноты поставленной задачи

Рассматриваемой задаче соответствует язык L – множество пар $<Sc, C>$ (здесь Sc – набор сценариев работы программы, C – натуральное число, записанное в унарной системе счисления), для которых существует автомат A, удовлетворяющий сценариям из Sc и число состояний которого равно C.

1.2.1. Используемые понятия теории сложности

Приведем понятия, используемые при проведении теоретической оценки сложности поставленной задачи. Данные понятия подробно изложены в [7].

Классом $DTIME(f(n))$ называется множество языков, для которых существует машина Тьюринга такая, что она всегда останавливается, и время ее работы не превосходит $f(n)$, где n – длина входа. Классом $NTIME(f(n))$, по аналогии с классом $DTIME$, называется класс языков (задач), для которых существует недетерминированная машина Тьюринга, такая, что она всегда останавливается, и время ее работы не превосходит $f(n)$, где n – длина входа.

Через понятия данных классов можно дать определение многим сложностным классам, в том числе используемым в рассмотрении P и NP. Класс P – класс языков (задач), разрешимых на детерминированной машине Тьюринга за полиномиальное время. Таким образом,

$$P = \bigcup_{i=0}^{\infty} DTIME\ (in^i)$$

В свою очередь, при разрешении языка из класса NP используется недетерминированная машина:

$$NP = \bigcup_{i=0}^{\infty} NTIME\ (in^i)$$

Альтернативным определением класса NP является определение на языке сертификатов. Говорят, что y является *сертификатом* принадлежности элемента x языку L, если существует полиномиальное отношение (верификатор) R, такое, что $R(x, y) = 1$ тогда и только тогда, когда x принадлежит L. Классом Σ_1 называется класс языков (задач) L, таких, что для каждого из них существует полиномиальный верификатор R, а также полином p, такие, что слово l принадлежит языку L тогда и только тогда, когда существует сертификат y, длина которого не превосходит заданного полинома p, и сертификат y удовлетворяет верификатору R. По теореме, приведенной в [7], классы Σ_1 и NP равны, следовательно, определение Σ_1 является альтернативным определением класса NP.

Введем понятие *сведение по Карпу*. Язык A сводится по Карпу к языку B, если существует функция $f(x)$ такая, что x принадлежит A тогда и только тогда, когда $f(x)$ принадлежит B. При этом сводящая функция должна быть вычислима за полиномиальное время от длины входа.

Теперь введем понятия классов NP-полных и NP-трудных задач. Язык L называется NP-трудным, если для любого языка M, принадлежащего NP, M сводится по Карпу к L. Язык L называется NP-полным, если является NP-полным и принадлежит классу NP.

1.2.2. Доказательство принадлежности поставленной задачи классу *NP*

Докажем принадлежность классу NP задачи построения управляющего автомата по сценариям работы. Для доказательства воспользуемся определением класса NP на языке сертификатов. Для исследуемой задачи сертификатом y для элемента $x = <Sc, C>$ является управляющий автомат, удовлетворяющий всем сценариям из Sc и число состояний которого равно C. Таким образом, верификатором $R(x, y)$ является программа, которая проверяет два утверждения.

Во-первых, для каждого сценария из множества *Sc* программа *R* проверяет, удовлетворяет ли ему автомат *y*. Следуя определению, введенному ранее, для каждого элемента сценария проверка, удовлетворяет ли элементу автомат, производится за $O(|y|)$ – в худшем случае будут рассмотрены все переходы автомата. Таким образом, для всех заданных сценариев время работы данного шага составит $O(|Sc|\cdot|y|)$, где $|Sc|$ – суммарная длина всех сценариев набора *Sc*.

Во-вторых, программа *R* проверяет то, что автомат состоит ровно из *C* состояний. В начале доказательства было оговорено, что число *C* задано в унарной системе счисления. Если бы это было иначе, то размер любого элемента *x* с пустым множеством *Sc* составлял бы $O(log\ |y|)$. Таким образом, размер сертификата *y* зависел бы экспоненциально от размера элемента *x*, что не позволило бы работать верификатору *R* полиномиальное время.

Таким образом, показано существование полиномиального отношения *R*, что доказывает принадлежность классу *NP* исследуемой задачи.

1.2.3. Доказательство принадлежности классу *NP*-трудных задач

Докажем принадлежность рассматриваемой задачи классу *NP*-трудных задач. Для этого докажем, что к поставленной задаче можно свести задачу из класса *NP*-трудных. В качестве такой задачи выберем задачу построения автомата-распознавателя с заданным числом состояний по заданному набору слов, которые автомат должен распознавать. Согласно работе [9] данная задача является *NP*-полной, а, следовательно, и *NP*-трудной.

Языком *M* данной задачи является множество троек $<S^+, S^-, C>$ таких, что существует конечный автомат-распознаватель, который принимает слова из словаря S^+, не принимает слова из словаря S^- и число его состояний равно *C*.

Согласно определению сведения по Карпу, язык *M* сводится к языку *L*, если существует такая функция $f(w)$, что *w* принадлежит *M* тогда и только тогда, когда элемент $f(w)$ принадлежит *L*.

Опишем сводящую функцию *f*, принимающую на вход элемент $w = <S^+, S^-, C>$ языка *M* и возвращающую элемент $f(w) = <Sc, C>$ языка *L* рассматриваемой в работе задачи. Множество сценариев *Sc* составляется следующим образом. Для каждого слова $a_1 \ldots a_n$ из словаря S^+ во множество *Sc* добавляется сценарий работы

$$<a_1, true, ()>, \ldots, <a_n, true, ()>, <\$, true, (accept)>,$$

где как «$» обозначен символ конца строки. Аналогично, для каждого слова из словаря S^- добавляется сценарий работы, последний элемент которого равен <$, *true*, (*reject*)>.

Докажем, что если $w = <S^+, S^-, C>$ принадлежит *M*, то $f(w) = <Sc, C>$ принадлежит языку *L* – если существует автомат-распознаватель *A* для элемента *w*, то можно построить управляющий автомат для $f(w)$. Для этого преобразуем автомат *A* следующим образом. Во-первых, каждый переход автомата-распознавателя по символу *c* преобразуем в переход управляющего автомата, помеченный тройкой $<a, true, ()>$ – событием *a*, тождественным охранным условием и пустой последовательностью выходных воздействий. Во-вторых, из каждого допускающего состояния добавим переход, помеченный тройкой <$, *true*, (*accept*)> и ведущий в это же состояние. При этом, разумеется, допускающие состояния преобразуются в обычные состояния управляющего автомата. В-третьих, для остальных состояний автомата аналогичным образом добавим переход, помеченный тройкой <$, *true*, (*reject*)>. Таким образом, легко видеть, что построенный управляющий автомат удовлетворяет сценариям из *Sc* и число его состояний равно *C*.

Докажем обратное. Если $f(w)$ принадлежит языку *L*, то *w* принадлежит языку *M* – если существует управляющий автомат *A'*, удовлетворяющий построенным сценариям из множества *Sc* и число его состояний равно *C*, то существует и автомат-распознаватель для элемента *w*. Преобразуем автомат *A'* в искомый автомат-распознаватель. Для этого выполним следующие действия:

- все переходы автомата, помеченные тройкой <*a*, *true*, ()>, преобразуем в переходы по символу *a;*
- переходы, помеченные <$, *true*, (*accept*)> удалим, при этом каждое состояние, из которого вел такой переход, сделаем допускающим;
- удалим остальные переходы.

Таким образом, полученный автомат-распознаватель по построению принимает слова из S^+, не принимает слова из S^- и число его состояний равно *C*.

Следовательно, доказано, что существует функция *f*(*w*) такая, что *w* принадлежит *M* тогда и только тогда, когда элемент *f*(*w*) принадлежит *L*. Следовательно, язык *M* сводится к языку *L*, что доказывает принадлежность задачи построения управляющего автомата классу *NP*-трудных задач.

1.3. Выводы по главе 1

1. Приведено формальное условие решаемой задачи – построение управляющего конечного автомата с заданным числом состояний *C* по заданному множеству сценариев работы *Sc*, которым автомат должен удовлетворять.
2. Приведены используемые понятия теории сложности. Определены классы *DTIME*, *NTIME*, *P*, *NP*, *NP*-трудных задач, *NP*-полных задач. Введено понятие сведения по Карпу.
3. Доказана принадлежность классу *NP* задачи построения управляющего автомата по сценариям работы. Для доказательства было использовано определение класса *NP* на языке сертификатов. Было показано существование верификатора – полиномиального отношения *R*.
4. Доказана принадлежность классу *NP*-трудных задач рассматриваемой задачи. Для этого было доказано, что к поставленной задаче можно свести задачу из класса *NP*-трудных. В качестве такой задачи была выбрана задача построения автомата-распознавателя с заданным числом состояний по заданному набору слов, которые автомат должен распознавать.
5. Таким образом, была приведена теоретическая оценка сложности рассматриваемой задачи – доказана ее *NP*-полнота.

2. РАЗРАБОТКА АЛГОРИТМА ПОСТРОЕНИЯ ДЕРЕВА СЦЕНАРИЕВ РАБОТЫ

В настоящем разделе приводится описание алгоритма построения дерева сценариев работы. Разработка алгоритма будет выполняться на основе методов теории графов.

2.1. Алгоритм построения дерева сценариев

Деревом сценариев назовем дерево, каждый переход которого помечен событием, булевой формулой и последовательностью выходных воздействий. Опишем алгоритм построения дерева сценариев по заданному множеству сценариев *Sc*.

Сначала дерево сценариев состоит из единственной вершины – корня дерева. Затем по очереди добавим в дерево все сценарии работы из *Sc*.

Для каждого из сценариев будем добавлять его элементы в дерево в порядке возрастания их номеров, начиная с первого. При этом будем хранить указатель на текущую вершину дерева *v* и номер *i* первого необработанного элемента сценария.

В начале процесса добавления *v* указывает на корень дерева сценариев, а $i = 1$. На каждом шаге проверяется существование исходящего из вершины *v* ребра, помеченного событием e_i и логической формулой, совпадающей с f_i как булевой функции. Если такое ребро не существует, то создается новая вершина дерева *u*, и в нее направляется ребро, помеченное тройкой $<e_i, f_i, A_i>$. После этого *u* становится текущей вершиной, а значение *i* увеличивается на единицу.

Если такое ребро существует, то производится сравнение последовательности A_i и последовательности выходных воздействий *A'*, которой помечено рассматриваемое ребро. Если $A_i = A'$, то текущей становится вершина, в которую ведет рассматриваемое ребро дерева, а значение *i* увеличивается на единицу.

Если же указанные последовательности не совпадают, то заданное множество сценариев *Sc* является противоречивым, поэтому работа алгоритма прерывается, и пользователю выводится соответствующее сообщение.

После завершения добавления всех сценариев в дерево производится проверка охранных условий. Для каждой вершины перебираются все пары исходящих из нее ребер. Если существует такая пара ребер, что они помечены одним и тем же событием, а их охранные условия имеют общий выполняющий набор значений входных переменных, то множество сценариев предполагает недетерминированное поведение. Поэтому работа алгоритма прерывается, и пользователю выводится соответствующее сообщение.

2.2. Примеры работы предложенного алгоритма

В настоящем разделе приведены примеры работы предложенного алгоритма построения дерева сценариев. Сначала будет рассмотрен небольшой набор сценариев, затем будет рассмотрен набор сценариев управления часами с будильником.

Набор сценариев работы должен быть в определенном смысле достаточно полным. Во-первых, если система со сложным поведением, для управления которой строится конечный автомат, имеет несколько режимов работы, то система сценариев должна содержать сценарии для каждого из этих режимов и для переходов между ними. Во-вторых, набор тестов в целом должен содержать все существующие в системе входные события и выходные воздействия.

2.2.1. Пример работы алгоритма на небольшом наборе сценариев

На рис. 2 приведен пример дерева сценариев, построенного по сценариям:

- $<A, \neg x, (z2)>$, $<A, x, (z1)>$, $<A, true, (z1,z1)>$;
- $<A, x, (z1)>$, $<B, true, (z1,z2)>$, $<A, x, (z1)>$;

- $\langle A, x, (z1)\rangle$, $\langle B, true, (z1,z2)\rangle$, $\langle A, \neg x, (z2)\rangle$, $\langle A, \neg x, (z2)\rangle$.

Данным сценариям удовлетворяет автомат, приведенный на рис. 1.

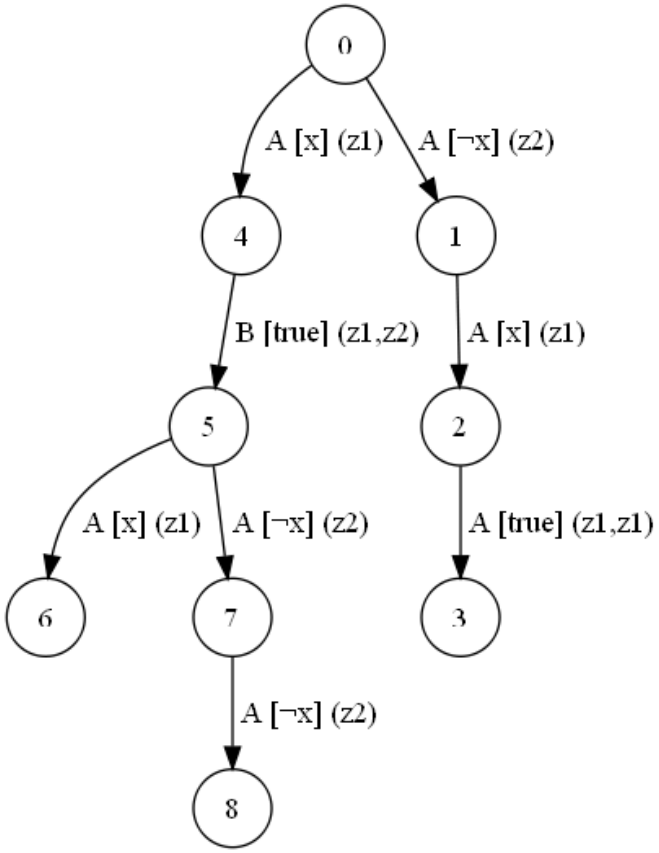

Рис. 2. Пример дерева сценариев

2.2.2. Пример работы алгоритма на сценариях управления часами с будильником

В настоящем разделе описывается пример применения предлагаемого алгоритма для построения дерева сценариев управления часами с будильником [4]. Сначала приводится описание модели часов с будильником, затем приводится система сценариев работы данных часов, затем приводится описание дерева сценариев, построенного по заданной системе сценариев работы.

2.2.2.1. Описание часов с будильником

Часы с будильником имеют три кнопки (рис. 3), которые предназначены для изменения режима их работы и для настройки текущего времени или времени срабатывания будильника.

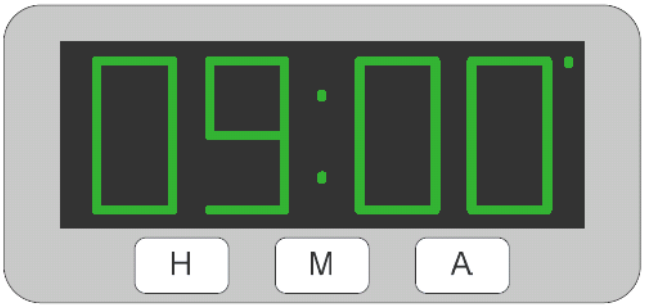

Рис. 3. Внешний вид часов с будильником

Если будильник выключен, то кнопки «H» и «M» предназначены, соответственно, для увеличения на единицу числа часов и минут в текущем времени. Кнопка «A» в этом режиме служит для перехода в режим настройки времени срабатывания будильника. В этом режиме кнопки «H» и «M» служат для увеличения на единицу числа часов и минут во времени срабатывания будильника. Нажатие кнопки «A» в этом режиме приводит к включению будильника. Он срабатывает, как только время срабатывания совпадает с текущим временем. Звонок автоматически выключается через минуту или может быть выключен нажатием кнопки «A», которая также выключает будильник. Кроме кнопок часы содержат таймер, который срабатывает каждую минуту – при каждом его срабатывании текущее время увеличивается на одну минуту.

Отметим, что рассматриваемые часы с будильником являются системой со сложным поведением, так как в ответ на одни и те же входные события (нажатия кнопок) в зависимости от режима работы генерируются различные выходные воздействия.

Эта система со сложным поведением имеет четыре входных события:

- H – нажата кнопка «H» на корпусе часов;
- M – нажата кнопка «M» на корпусе часов;
- A – нажата кнопка «A» на корпусе часов;
- T – сработал таймер.

Она также содержит две входные переменные:

- $x1$ – верно ли, что время срабатывания будильника совпадает с текущим временем?
- $x2$ – верно ли, что текущее время превышает время срабатывания будильника ровно на одну минуту?

Кроме этого эта система имеет семь выходных воздействий:

- $z1$ – увеличить число часов текущего времени;
- $z2$ – увеличить число минут часов текущего времени;
- $z3$ – увеличить число часов времени срабатывания будильника;
- $z4$ – увеличить число минут времени срабатывания будильника;
- $z5$ – прибавить минуту к текущему времени;
- $z6$ – включить звонок будильника;
- $z7$ – выключить звонок будильника.

Поведение часов с будильником может быть описано графом переходов конечного автомата, который приведен в [4] и построен вручную (рис. 4).

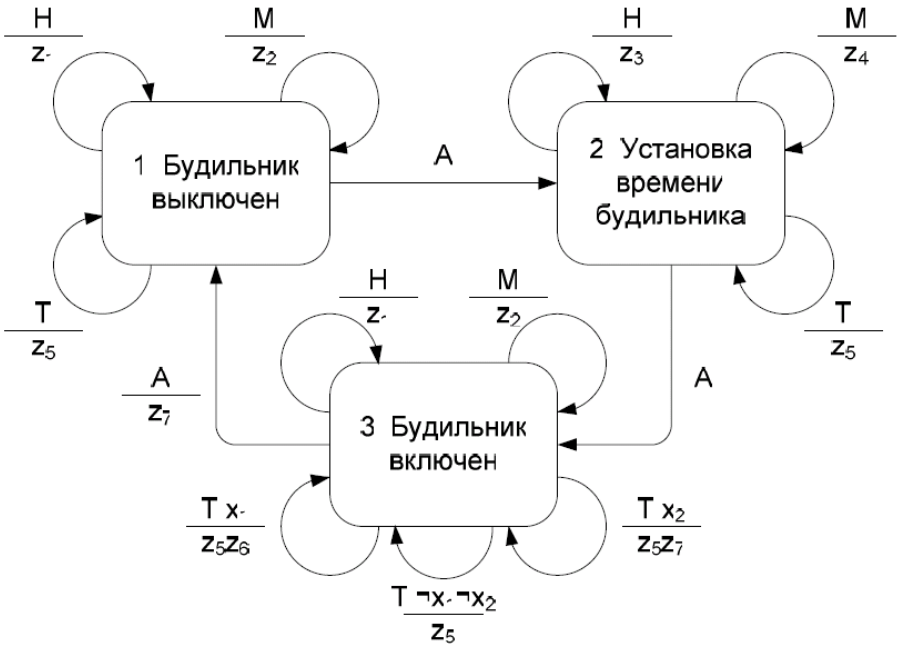

Рис. 4. Граф переходов конечного автомата управления часами с будильником

Начальным состоянием этого автомата является состояние «1. Будильник выключен».

2.2.2.2. Система сценариев работы автомата управления часами

В систему сценариев работы автомата управления включены сценарии, описывающие его работу во всех трех режимах. Сценарии работы, описывающие поведение часов с будильником в режиме работы «Будильник выключен» приведены в табл. 1.

Таблица 1. Сценарии работы для режима работы «Будильник выключен»

Сценарий работы	Комментарий
<*T*, *true*, (z5)>, <*T*, *true*, (z5)>, <*T*, *true*, (z5)>, <*T*, *true*, (z5)>	Описывает обработку часами события «Сработал таймер». При возникновении этого события текущее время должно быть увеличено на минуту.
<*H*, *true*, (z1)>, <*H*, *true*, (z1)>, <*H*, *true*, (z1)>, <*H*, *true*, (z1)>	Описывает обработку часами нажатия кнопки «H». При нажатии на эту кнопку число часов в текущем времени должно быть увеличено на единицу.
<*M*, *true*, (z2)>, <*M*, *true*, (z2)>, <*M*, *true*, (z2)>, <*M*, *true*, (z2)>	Описывает обработку часами нажатия кнопки «M». При нажатии на эту кнопку число минут в текущем времени должно быть увеличено на единицу.
<*T*, *true*, (z5)>, <*M*, *true*, (z2)>, <*H*, *true*, (z1)>, <*T*, *true*, (z5)>, <*T*, *true*, (z5)>, <*T*, *true*, (z5)>, <*M*, *true*, (z2)>, <*T*, *true*, (z5)>, <*H*, *true*, (z1)>, <*H*, *true*, (z1)>, <*T*, *true*, (z5)>, <*M*, *true*, (z2)>	Описывает обработку событий H, M и T в режиме «Будильник выключен».
<*A*, *true*, ()>, <*A*, *true*, ()>, <*A*, *true*, (z7)>, <*T*, *true*, (z5)>, <*T*, *true*, (z5)>, <*T*, *true*, (z5)>, <*T*, *true*, (z5)>	После трех нажатий кнопки «A» часы должны находиться в режиме «Будильник выключен». Аналог первого сценария работы.

<*A*, *true*, ()>, <*A*, *true*, ()>, <*A*, *true*, (z7)>, <*H*, *true*, (z1)>, <*H*, *true*, (z1)>, <*H*, *true*, (z1)>, <*H*, *true*, (z1)>	После трех нажатий кнопки «A» часы должны находиться в режиме «Будильник выключен». Аналог второго сценария работы.
<*A*, *true*, ()>, <*A*, *true*, ()>, <*A*, *true*, (z7)>, <*M*, *true*, (z2)>, <*M*, *true*, (z2)>, <*M*, *true*, (z2)>, <*M*, *true*, (z2)>	После трех нажатий кнопки «A» часы должны находиться в режиме «Будильник выключен». Аналог третьего сценария работы.

Сценарии работы, описывающие поведение часов с будильником в режиме работы «Настройка будильника» приведены в табл. 2.

Таблица 2. Сценарии работы для режима работы «Настройка будильника»

Сценарий работы	Комментарий
<*A*, *true*, ()>, <*T*, *true*, (z5)>, <*T*, *true*, (z5)>, <*T*, *true*, (z5)>, <*T*, *true*, (z5)>	Описывает обработку часами события «Сработал таймер». При возникновении этого события текущее время должно быть увеличено на минуту.
<*A*, *true*, ()>, <*H*, *true*, (z3)>, <*H*, *true*, (z3)>, <*H*, *true*, (z3)>, <*H*, *true*, (z3)>	Описывает обработку часами нажатия кнопки «H». При нажатии на эту кнопку число часов во времени срабатывания будильника должно быть увеличено на единицу.
<*A*, *true*, ()>, <*M*, *true*, (z4)>, <*M*, *true*, (z4)>, <*M*, *true*, (z4)>, <*M*, *true*, (z4)>	Описывает обработку часами нажатия кнопки «M». При нажатии на эту кнопку число минут во времени срабатывания будильника должно быть увеличено на единицу.
<*A*, *true*, ()>, <*T*, *true*, (z5)>, <*M*, *true*, (z4)>, <*H*, *true*, (z3)>, <*T*, *true*, (z5)>, <*T*, *true*, (z5)>, <*T*, *true*, (z5)>, <*M*, *true*, (z4)>, <*T*, *true*, (z5)>, <*H*, *true*, (z3)>, <*H*, *true*, (z3)>, <*T*, *true*, (z5)>, <*M*, *true*, (z4)>	Описывает обработку событий H, M и T в режиме «Настройка будильника».
<*A*, *true*, ()>, <*A*, *true*, ()>, <*A*, *true*, (z7)>, <*A*, *true*, ()>, <*T*, *true*, (z5)>	После четырех нажатий кнопки «A» часы должны находиться в режиме «Настройка будильника». Описывается обработка срабатывания таймера.
<*A*, *true*, ()>, <*A*, *true*, ()>, <*A*, *true*, (z7)>, <*A*, *true*, ()>, <*T*, *true*, (z5)>, <*T*, *true*, (z5)>, <*T*, *true*, (z5)>, <*T*, *true*, (z5)>	После четырех нажатий кнопки «A» часы должны находиться в режиме «Настройка будильника». Описывается обработка нескольких срабатываний таймера.
<*A*, *true*, ()>, <*A*, *true*, ()>, <*A*, *true*, (z7)>, <*A*, *true*, ()>, <*H*, *true*, (z3)>	После четырех нажатий кнопки «A» часы должны находиться в режиме «Настройка будильника». Описывается обработка нажатия кнопки «H».
<*A*, *true*, ()>, <*A*, *true*, ()>, <*A*, *true*, (z7)>, <*A*, *true*, ()>, <*H*, *true*, (z3)>, <*H*, *true*, (z3)>, <*H*, *true*, (z3)>, <*H*, *true*, (z3)>	После четырех нажатий кнопки «A» часы должны находиться в режиме «Настройка будильника». Описывается обработка нескольких нажатий кнопки «H».
<*A*, *true*, ()>, <*A*, *true*, ()>, <*A*, *true*, (z7)>, <*A*, *true*, ()>, <*M*, *true*, (z4)>	После четырех нажатий кнопки «A» часы должны находиться в режиме «Настройка будильника». Описывается обработка нажатия кнопки «M».
<*A*, *true*, ()>, <*A*, *true*, ()>, <*A*, *true*, (z7)>, <*A*, *true*, ()>, <*M*, *true*, (z4)>, <*M*, *true*, (z4)>,	После четырех нажатий кнопки «A» часы должны находиться в режиме «Настройка будильника».

<*M*, *true*, (*z*4)>, <*M*, *true*, (*z*4)>	Описывается обработка нескольких нажатий кнопки «M».

Сценарии работы, описывающие поведение часов с будильником в режиме работы «Будильник включен» приведены в табл. 3.

Таблица 3. Сценарии работы для режима работы «Будильник включен»

Сценарий	Комментарий
<*A*, *true*, ()>, <*A*, *true*, ()>, <*H*, *true*, (*z*1)>, <*H*, *true*, (*z*1)>, <*H*, *true*, (*z*1)>, <*H*, *true*, (*z*1)>	Описывает обработку часами нажатия кнопки «H». При нажатии на эту кнопку число часов в текущем времени должно быть увеличено на единицу.
<*A*, *true*, ()>, <*A*, *true*, ()>, <*M*, *true*, (*z*2)>, <*M*, *true*, (*z*2)>, <*M*, *true*, (*z*2)>, <*M*, *true*, (*z*2)>	Описывает обработку часами нажатия кнопки «M». При нажатии на эту кнопку число минут в текущем времени должно быть увеличено на единицу.
<*A*, *true*, ()>, <*A*, *true*, ()>, <*T*, ¬*x*1&¬*x*2, (*z*5)>	Описывает обработку часами события «Сработал таймер» при условии, что текущее время не совпадает со временем срабатывания будильника или со временем срабатывания будильника, увеличенными на минуту. Текущее время должно быть увеличено на минуту.
<*A*, *true*, ()>, <*A*, *true*, ()>, <*T*, ¬*x*1&¬*x*2, (*z*5)>, <*T*, ¬*x*1&¬*x*2, (*z*5)>, <*T*, ¬*x*1&¬*x*2, (*z*5)>	Расширение предыдущего сценария.
<*A*, *true*, ()>, <*A*, *true*, ()>, <*T*, ¬*x*1&¬*x*2, (*z*5)>, <*T*, *x*1&¬*x*2, (*z*5, *z*6)>, <*T*, *x*2&¬*x*1, (*z*5, *z*7)>	Описывает обработку события «Сработал таймер» при различных значениях входных переменных.
<*A*, *true*, ()>, <*A*, *true*, ()>, <*T*, ¬*x*1&¬*x*2, (*z*5)>, <*T*, *x*1&¬*x*2, (*z*5, *z*6)>	Префикс предыдущего сценария.
<*A*, *true*, ()>, <*A*, *true*, ()>, <*T*, ¬*x*1&¬*x*2, (*z*5)>, <*T*, *x*1&¬*x*2, (*z*5, *z*6)>, <*T*, *x*2&¬*x*1, (*z*5, *z*7)>, <*H*, *true*, (*z*1)>	Описывает обработку события «Сработал таймер» при различных значениях входных переменных, а также обработку нажатие кнопки «H».
<*A*, *true*, ()>, <*A*, *true*, ()>, <*T*, ¬*x*1&¬*x*2, (*z*5)>, <*T*, *x*1&¬*x*2, (*z*5, *z*6)>, <*T*, *x*2&¬*x*1, (*z*5, *z*7)>, <*M*, *true*, (*z*2)>	Описывает обработку события «Сработал таймер» при различных значениях входных переменных, а также обработку нажатие кнопки «M».
<*A*, *true*, ()>, <*A*, *true*, ()>, <*T*, ¬*x*1&¬*x*2, (*z*5)>, <*T*, *x*1&¬*x*2, (*z*5, *z*6)>, <*T*, *x*2&¬*x*1, (*z*5, *z*7)>, <*T*, ¬*x*1&¬*x*2, (*z*5)>	Описывает обработку события «Сработал таймер» при различных значениях входных переменных.
<*A*, *true*, ()>, <*A*, *true*, ()>, <*T*, *x*1&¬*x*2, (*z*5, *z*6)>	Описывает обработку события «Сработал таймер» при условии, что текущее время совпадает со временем срабатывания будильника.

<*A*, *true*, ()>, <*A*, *true*, ()>, <*T*, *x*2&¬*x*1, (*z*5, *z*7)>	Описывает обработку события «Сработал таймер» при условии, что текущее время превышает на минуту время срабатывания будильника.
<*A*, *true*, ()>, <*A*, *true*, ()>, <*T*, *x*1&¬*x*2, (*z*5, *z*6)>, <*T*, *x*2&¬*x*1, (*z*5, *z*7)>	Объединяет два предыдущих сценария.
<*A*, *true*, ()>, <*A*, *true*, ()>, <*T*, ¬*x*1&¬*x*2, (*z*5)>, <*M*, *true*, (*z*2)>, <*H*, *true*, (*z*1)>, <*T*, ¬*x*1&¬*x*2, (*z*5)>, <*T*, ¬*x*1&¬*x*2, (*z*5)>, <*T*, ¬*x*1&¬*x*2, (*z*5)>, <*M*, *true*, (*z*2)>, <*T*, ¬*x*1&¬*x*2, (*z*5)>, <*H*, *true*, (*z*1)>, <*H*, *true*, (*z*1)>, <*T*, ¬*x*1&¬*x*2, (*z*5)>, <*M*, *true*, (*z*2)>	Описывает обработку событий H, M и T в режиме «Будильник включен».

Кроме сценариев работы, описывающих поведение автомата в каждом из трех режимов, в набор сценариев включены сценарии, описывающие поведение автомата сразу в нескольких режимах. Эти сценарии приведены в табл. 4.

Таблица 4. Сценарии работы, описывающие поведение автомата в нескольких режимах

Сценарий	Комментарий
<*A*, *true*, ()>, <*A*, *true*, ()>, <*T*, ¬*x*1&¬*x*2, (*z*5)>, <*T*, *x*1&¬*x*2, (*z*5, *z*6)>, <*T*, *x*2&¬*x*1, (*z*5, *z*7)>, <*A*, *true*, (*z*7)>, <*H*, *true*, (*z*1)>	Описывает поведение часов с будильником в двух режимах: «Будильник включен» и «Будильник выключен».
<*A*, *true*, ()>, <*A*, *true*, ()>, <*T*, ¬*x*1&¬*x*2, (*z*5)>, <*T*, *x*1&¬*x*2, (*z*5, *z*6)>, <*T*, *x*2&¬*x*1, (*z*5, *z*7)>, <*A*, *true*, (*z*7)>, <*M*, *true*, (*z*2)>	Описывает поведение часов с будильником в двух режимах: «Будильник включен» и «Будильник выключен».
<*A*, *true*, ()>, <*A*, *true*, ()>, <*T*, ¬*x*1&¬*x*2, (*z*5)>, <*T*, *x*1&¬*x*2, (*z*5, *z*6)>, <*T*, *x*2&¬*x*1, (*z*5, *z*7)>, <*A*, *true*, (*z*7)>, <*T*, *true*, (*z*5)>	Описывает поведение часов с будильником в двух режимах: «Будильник включен» и «Будильник выключен».
<*A*, *true*, ()>, <*A*, *true*, ()>, <*T*, ¬*x*1&¬*x*2, (*z*5)>, <*T*, *x*1&¬*x*2, (*z*5, *z*6)>, <*T*, *x*2&¬*x*1, (*z*5, *z*7)>, <*A*, *true*, (*z*7)>, <*A*, *true*, ()>, <*H*, *true*, (*z*3)>	Описывает поведение часов с будильником в двух режимах: «Будильник включен» и «Настройка будильника».
<*A*, *true*, ()>, <*A*, *true*, ()>, <*T*, ¬*x*1&¬*x*2, (*z*5)>, <*T*, *x*1&¬*x*2, (*z*5, *z*6)>, <*T*, *x*2&¬*x*1, (*z*5, *z*7)>, <*A*, *true*, (*z*7)>, <*A*, *true*, ()>, <*M*, *true*, (*z*4)>	Описывает поведение часов с будильником в двух режимах: «Будильник включен» и «Настройка будильника».
<*A*, *true*, ()>, <*A*, *true*, ()>, <*T*, ¬*x*1&¬*x*2, (*z*5)>, <*T*, *x*1&¬*x*2, (*z*5, *z*6)>, <*T*, *x*2&¬*x*1, (*z*5, *z*7)>, <*A*, *true*, (*z*7)>, <*A*, *true*, ()>, <*T*, *true*, (*z*5)>	Описывает поведение часов с будильником в двух режимах: «Будильник включен» и «Настройка будильника».
<*A*, *true*, ()>, <*A*, *true*, ()>, <*T*, ¬*x*1&¬*x*2, (*z*5)>, <*T*, *x*1&¬*x*2, (*z*5, *z*6)>, <*A*, *true*, (*z*7)>, <*A*, *true*, ()>, <*T*, *true*, (*z*5)>	Описывает поведение часов с будильником в двух режимах: «Будильник включен» и «Настройка будильника».

2.2.2.3. Пример работы алгоритма построения дерева сценариев

По заданной системе сценариев работы построим дерево сценариев. Так как данное дерево достаточно велико, разобьем для удобства восприятия его на части, которые приведены рис. 5 – рис. 8. Каждая из частей соответствует одной из таблиц, приведенных выше. Также для удобства вершины дерева приведены без своих номеров в дереве.

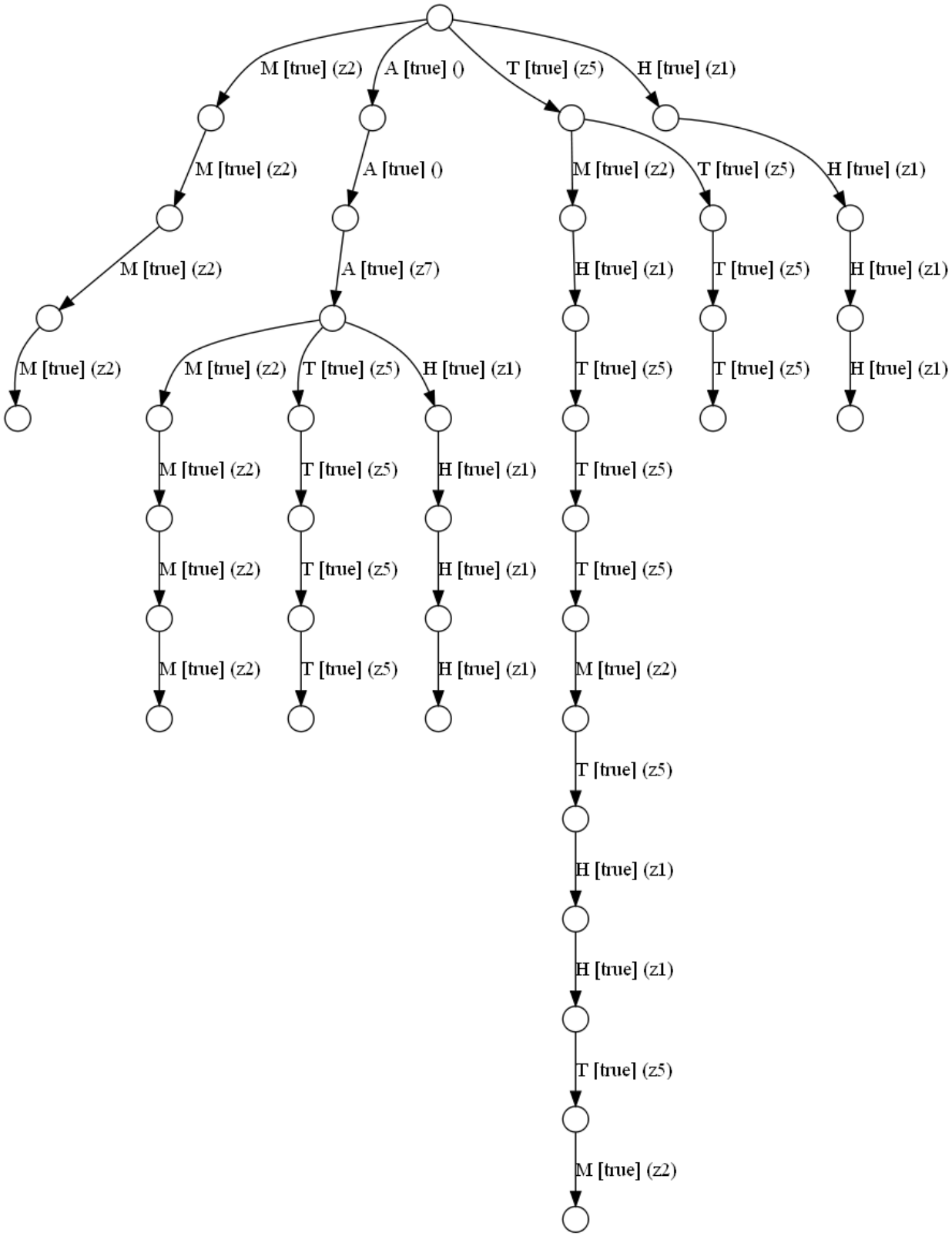

Рис. 5. Часть дерева сценариев, соответствующая сценариям работы, приведенным в табл. 1

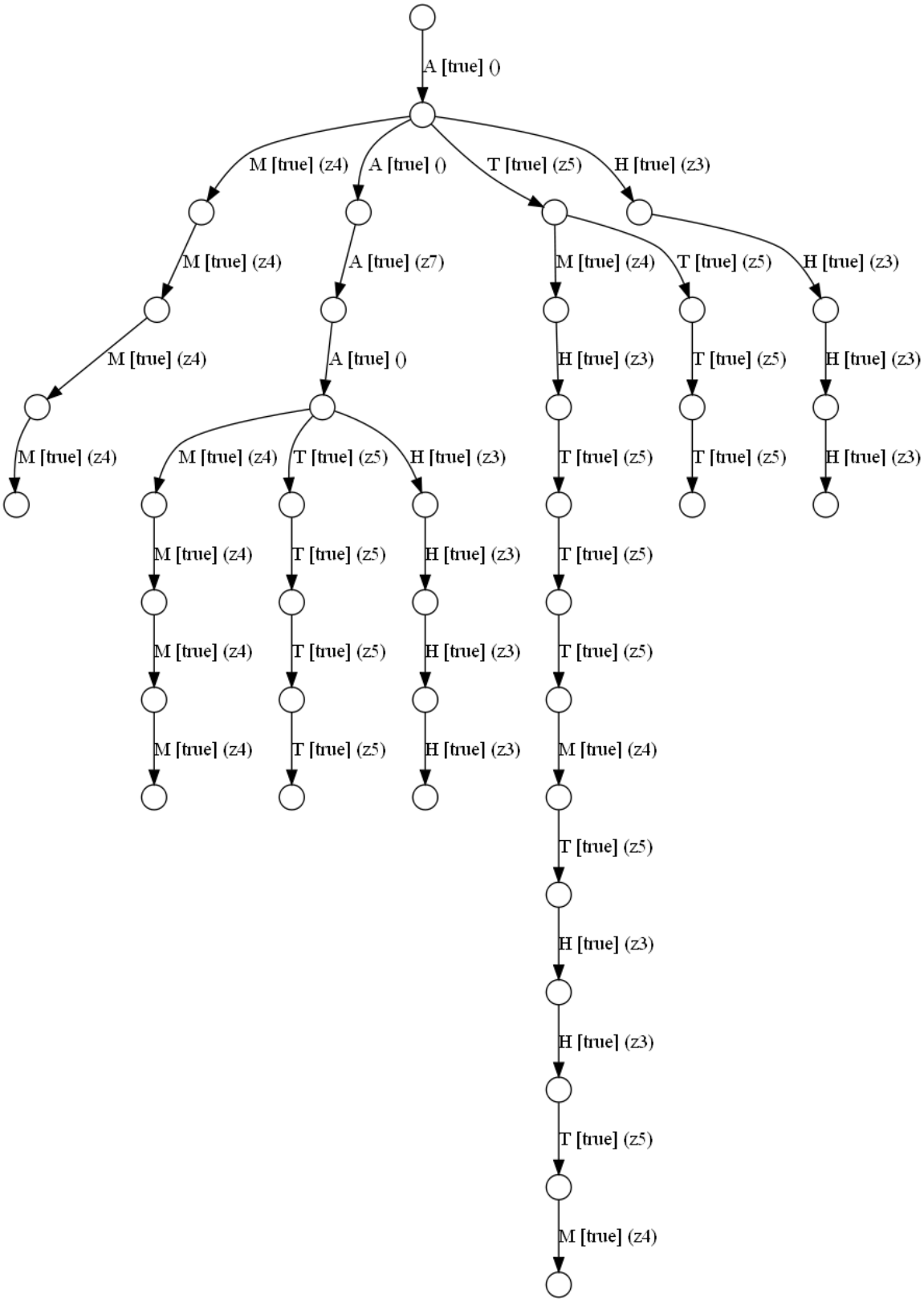

Рис. 6. Часть дерева сценариев, соответствующая сценариям работы, приведенным в табл. 2

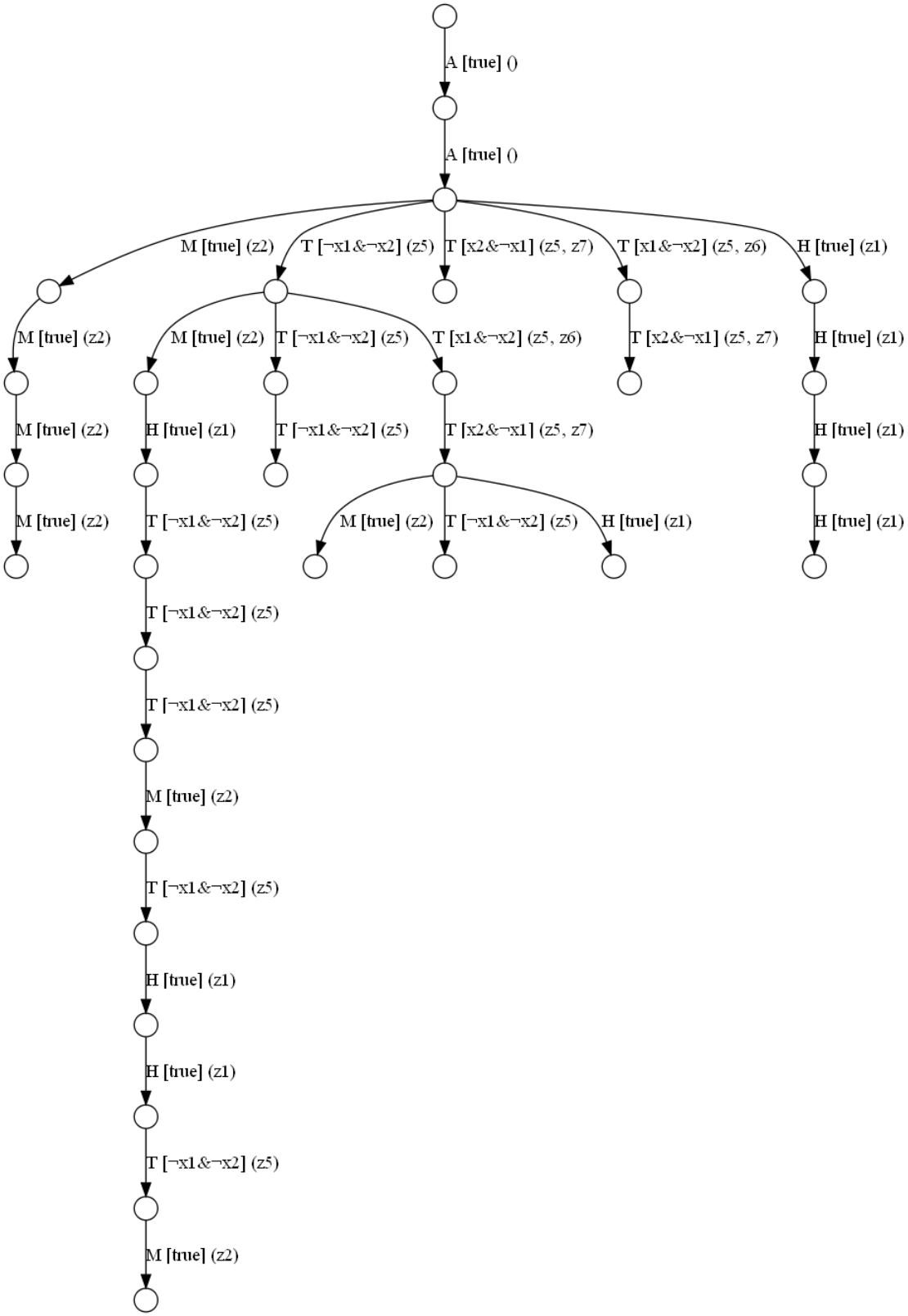

Рис. 7. Часть дерева сценариев, соответствующая сценариям работы, приведенным в табл. 3

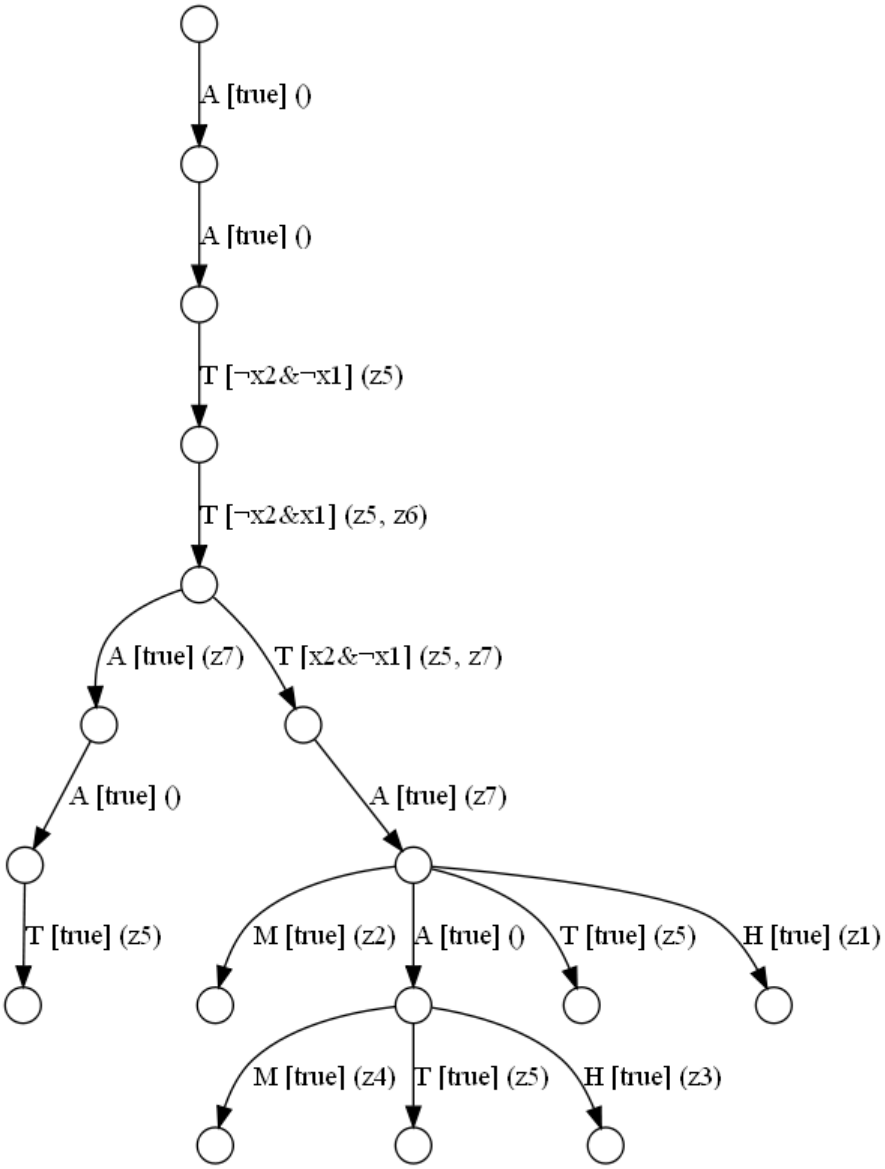

Рис. 8. Часть дерева сценариев, соответствующая сценариям работы, приведенным в табл. 4

2.3. Выводы по главе 2

1. Разработан алгоритм построения дерева сценариев работы. Разработка алгоритма осуществлялась на основе методов теории графов.
2. Приведен небольшой пример работы разработанного алгоритма – дерево сценариев строилось из трех сценариев работы.
3. Приведен пример работы разработанного алгоритма на системе сценариев работы автомата управления часами с будильником. Приведено описание данных часов и автомата управления, построенного вручную. Описана система из 38 сценариев работы.

3. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ АЛГОРИТМА ПОСТРОЕНИЯ ДЕРЕВА СЦЕНАРИЕВ РАБОТЫ

Программная реализация разработанного алгоритма построения дерева сценариев работы выполнялась на языке программирования *Java*.

В настоящем разделе приведены основные классы, используемые при реализации разрабатываемого метода машинного обучения, а также реализация разработанного алгоритма с использованием данных классов. Классы и связи между ними представлены на схеме (рис. 9).

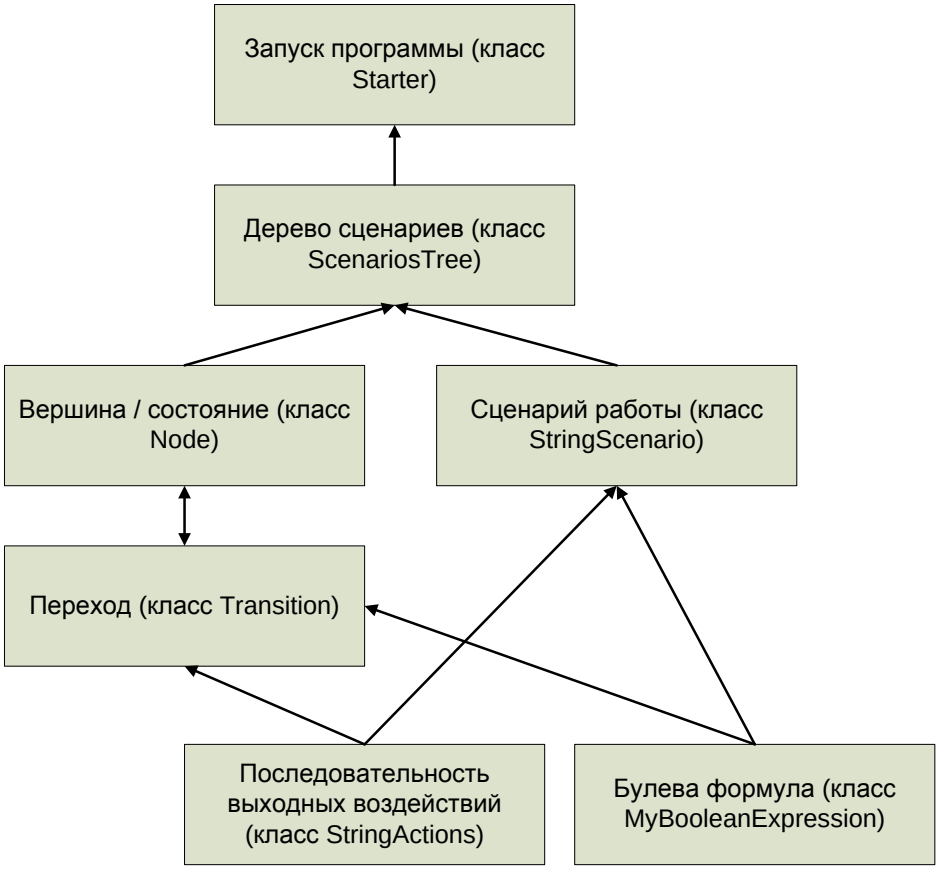

Рис. 9. Структура программной реализации алгоритма

3.1. Базовые классы реализации

В настоящем разделе приведено описание базовых классов, необходимых при реализации алгоритма построения дерева сценариев. Такими классами и соответствующими им сущностями являются:

- `StringActions` – последовательность выходных воздействий;
- `MyBooleanExpression` – булева формула;
- `Transition` – переход дерева сценариев;
- `Node` – вершина дерева сценариев;
- `StringScenario` – сценарий работы программы.

3.1.1. Программная реализация сущности «Последовательность выходных воздействий»

Класс `StringActions` содержит программную реализацию сущности «Последовательность выходных воздействий». Каждое выходное воздействие описывается элементом типа `String`. Единственное поле класса описывает данную последовательность.

```
private String[] actions;
```

Приведем реализацию конструктора данного класса, используемого при реализации алгоритма. Экземпляр класса `StringActions` конструируется от строки `str` типа `String`. Данная строка разбивается на выходные воздействия по символу «,».

```
public StringActions(String str) {
  str = str.trim();
  if (str.equals("")) {
    this.actions = new String[0];
  } else {
    this.actions = str.split(",");
  }
  for (int i = 0; i < actions.length; i++) {
    this.actions[i] = this.actions[i].trim();
  }
}
```

Полная программная реализация класса `StringActions` приведена в приложении 1.

3.1.2. Программная реализация сущности «Булева формула»

Класс `MyBooleanExpression` содержит программную реализацию сущности «Булева формула». Данной булевой формулой помечены переходы дерева сценариев. Данный класс использует классы `BooleanExpression` и `TruthTable` программного средства Boolean Expression Solver [9].

Поля класса описывают строковое представление `repr` формулы, набор переменных `variables` формулы, таблицу истинности `truthTable` формулы:

```
private String repr;
private String[] variables;
private Map<Map<String, Boolean>, Boolean> truthTable;
```

Приведем реализацию конструктора данного класса. Экземпляр класса `MyBooleanExpression` создается из строки `expression`. При его создании может произойти ошибка типа `ParseException`, что означает, что формула задана некорректно.

```
public MyBooleanExpression(String expression) throws ParseException {
  expression = expression.replaceAll(" ", "").replaceAll("!", "~");
  this.repr = expression;
  Pattern pattern = Pattern.compile("[()~&|=>+]");
  String[] vars = pattern.split(expression);
```

```
    HashSet<String> varsSet = new HashSet<String>(Arrays.asList(vars));
    varsSet.removeAll(Arrays.asList(new String[]{"", "1", "0"}));
    this.variables = varsSet.toArray(new String[0]);

    String shortExpr = new String(expression);
    for (int i = 0; i < variables.length; i++) {
      shortExpr = shortExpr.replaceAll(variables[i],
                                "" + (char)('a' + i));
    }

    BooleanExpression booleanExpression =
            new BooleanExpression(shortExpr);
    Map<Map<String, Boolean>, Map<BooleanExpression, Boolean>> truthTable
            = new TruthTable(booleanExpression).getResults();
    this.truthTable = new HashMap<Map<String,Boolean>, Boolean>();

    for (Map<String, Boolean> map : truthTable.keySet()) {
      Map<BooleanExpression, Boolean> booleanMap = truthTable.get(map);
      boolean val = booleanMap.containsValue(true);
      this.truthTable.put(map, val);
    }
  }
```

Еще одним важным методом данного класса является метод `hasSolutionWith`, который принимает другой экземпляр класса `MyBooleanExpression` и определяет, есть ли у формул общий выполняющий набор значений переменных. Для того чтобы не вычислять повторно значение функции, результаты вычислений сохраняются в поле рассматриваемого класса `hasSolutionWithRes` типа `Map<MyBooleanExpression, Boolean>`.

```
  private Map<MyBooleanExpression, Boolean> hasSolutionWithRes;

  public boolean hasSolutionWith(MyBooleanExpression other) {
    if (hasSolutionWithRes == null) {
      hasSolutionWithRes = new HashMap<MyBooleanExpression, Boolean>();
    }

    if (hasSolutionWithRes.containsKey(other)) {
      return hasSolutionWithRes.get(other);
    }

    MyBooleanExpression e =
        new MyBooleanExpression("(" + repr + ")&(" + other.repr + ")");
    boolean res = e.hasSolution();
    hasSolutionWithRes.put(other, res);
    return res;
  }
```

Полная программная реализация класса `MyBooleanExpression` приведена в приложении 2.

3.1.3. Программная реализация сущности «Переход дерева сценариев»

Класс `Transition` содержит программную реализацию сущности «Переход дерева сценариев». Опишем поля данного класса.

Поле `event` описывает входное событие, которым помечен переход. В рамках настоящей работы будем считать, что входные события являются строками типа `String`.

```
private String event;
```

Поле `expr` описывает булеву формулу, которой помечен переход. Формула имеет тип `MyBooleanExpression`, описание которого приведено выше.

```
private MyBooleanExpression expr;
```

Поле `actions` описывает последовательность выходных воздействий, которой помечен переход. Данная последовательность имеет тип `StringActions`, описание которого приведено выше.

```
private StringActions actions;
```

Наконец, поля `src` и `dst` описывают вершины дерева сценариев из которого и в которое ведет данный переход, соответственно. Тип `Node` данных вершин будет описан далее.

```
private Node src;
private Node dst;
```

Полная программная реализация класса `Transition` приведена в приложении 3.

3.1.4. Программная реализация сущности «Вершина дерева сценариев»

Класс `Node` содержит программную реализацию сущности «Вершина дерева сценариев». Поля класса описывают номер `number` вершины в дереве и переходы `transitions`, ведущие из данной вершины. Переходы хранятся в структуре типа `Map<String, Transition>`, которая сопоставляет переход строковому представлению входного события и булевой формулы, которого достаточно для идентификации перехода.

```
private int number;
private Map<String, Transition> transitions;
```

При таком способе хранения методы добавления `addTransition` и взятия `getTransition` перехода можно реализовать следующим образом:

```
public void addTransition(String event, MyBooleanExpression expr,
                  StringActions actions, Node dst) {
  Transition transition =
                  new Transition(this, dst, event, expr, actions);
```

```
  transitions.put(event + "[" + expr.toString() + "]", transition);
}

public Transition getTransition(String event,
                  MyBooleanExpression expr) {
  return transitions.get(event + "[" + expr.toString() + "]");
}
```

Полная программная реализация класса `Node` приведена в приложении 4.

3.1.5. Программная реализация сущности «Сценарий работы программы»

Класс `StringScenario` содержит программную реализацию сущности «Сценарий работы программы». Поля класса описывают последовательность троек $<e_i, f_i, A_i>$ данного сценария – поля описывают последовательность `events` входных событий сценария работы, последовательность `expressions` булевых формул, последовательность `actions` последовательностей выходных действий.

```
private String[] events;
private MyBooleanExpression[] expressions;
private StringActions[] actions;
```

Основной деталью реализации данного класса является наличие статического поля `exprList` типа `List<MyBooleanExpression>`, общего для всех экземпляров класса.

```
private static List<MyBooleanExpression> exprList =
          new ArrayList<MyBooleanExpression>();
```

Данное поле используется для хранения всех булевых формул (типа `MyBooleanExpression`), которые встречались в обрабатываемом множестве сценариев. При этом хранятся только булевы формулы с уникальными таблицами истинности. Для этого при обработке очередной булевой формулы производится сравнение ее таблицы истинности с таблицами истинности уже добавленных формул. Если такая формула уже встречалась, то в дальнейшем будем на нее ссылаться. Если нет, то добавим в список `exprList` новую формулу. Такой способ обработки в дальнейшем позволяет проверять равенство двух булевых формул типа `MyBooleanExpression` оператором «==».

Описанный механизм обработки реализован в конструкторе класса `StringScenario`, принимающем на вход строку `input` входных событий с охранными условиями и строку `output`, содержащую соответствующие последовательности выходных воздействий.

```
public StringScenario(String input, String output)
     throws ParseException {
  ...
    MyBooleanExpression eq = null;
    for (MyBooleanExpression e2 : exprList) {
      if (newExpr.equals(e2)) {
        eq = e2;
        break;
```

```
        }
      }
      if (eq == null) {
        exprList.add(newExpr);
        eq = newExpr;
      }
      this.expressions[i] = eq;
    ...
  }
```

Здесь и в дальнейшем под «…» будем подразумевать несущественные элементы реализации. Полная программная реализация класса `StringScenario` приведена в приложении 5.

3.2. Реализация хранения и алгоритма построения дерева сценариев работы

В настоящем разделе приведено описание класса `ScenariosTree`, реализующего сущность «Дерево сценариев». Данный класс позволяет хранить и строить дерево сценариев работы по заданному множеству сценариев. При этом будут использоваться базовые классы, описанные выше.

3.2.1. Хранение дерева сценариев

Поля класса `ScenariosTree` описывают корень дерева `root` типа `Node` и список всех вершин дерева сценариев `nodes` типа `List<Node>`.

```
private Node root;
private List<Node> nodes;
```

Данный класс имеет единственный конструктор, по умолчанию создающий дерево сценариев из единственной вершины – корня дерева.

```
public ScenariosTree() {
  this.root = new Node(0);
  this.nodes = new ArrayList<Node>();
  this.nodes.add(root);
}
```

Опишем методы, предоставляющие доступ к полям класса. Метод `getRoot()` возвращает ссылку на корень дерева сценариев. Метод `getNodes()` возвращает ссылку на список всех вершин дерева. Метод `nodesCount()` возвращает число вершин данного дерева.

```
public Node getRoot();
public List<Node> getNodes();
public int nodesCount();
```

3.2.2. Построение дерева сценариев

Опишем методы класса `ScenariosTree`, предоставляющие пользователю возможность построения дерева сценариев.

Сначала опишем вспомогательные методы класса. Скрытый метод `addTransition` позволяет добавить переход, исходящий из вершины `src` и помеченный событием `event`, булевой

формулой expr и последовательностью выходных воздействий actions. Если из вершины src уже ведет переход, помеченный событием event, булевой формулой expr и последовательностью выходных воздействий, не совпадающей с actions, генерируется ошибка типа ParseException. Если же такого перехода не существует, создается новая вершина дерева dst, в которое направляется новый переход.

```
private void addTransition(Node src, String event, MyBooleanExpression
            expr, StringActions actions) throws ParseException {
  if (src.hasTransition(event, expr)) {
    Transition t = src.getTransition(event, expr);
    if (!t.getActions().equals(actions)) {
      throw new ParseException("bad transition add in node " +
       src.getNumber() + ": " + t.getActions() + " != " + actions, 0);
    }
  } else {
    Node dst = new Node(nodes.size());
    nodes.add(dst);
    src.addTransition(event, expr, actions, dst);
  }
}
```

Скрытый метод addScenario класса ScenariosTree позволяет добавить сценарий работы в дерево сценариев. При этом сначала конструируется сценарий работы scenario по заданным параметрам – строкам input и output, затем элементы сценария последовательно добавляются в дерево сценариев, используя метод addTransition, описанный выше. Возможные ошибки типа ParseException метод addScenario не обрабатывает, а передает дальше.

```
private void addScenario(String input, String output)
              throws ParseException {
  StringScenario scenario = new StringScenario(input, output);
  Node node = root;
  for (int i = 0; i < scenario.size(); i++) {
    addTransition(node, scenario.getEvent(i), scenario.getExpr(i),
              scenario.getActions(i));
    node = node.getDst(scenario.getEvent(i), scenario.getExpr(i));
  }
}
```

Пользователю предоставляется возможность загрузить дерево сценариев из файла, который содержит множество сценариев работы. Для этого в классе ScenariosTree реализован метод load, принимающий на вход путь filepath до данного файла. Данный метод, игнорируя пустые строки, считывает строки с входными событиями (с охранными условиями) и выходными воздействиями, передавая их методу addScenario, описанному выше. При этом ошибка типа ParseException не обрабатывается, а передается дальше. Так как файла по пути filepath может не быть, метод генерирует ошибку типа FileNotFoundException.

```
public void load(String filepath) throws
```

```
                FileNotFoundException, ParseException {
  Scanner in = new Scanner(new File(filepath));

  String inp = "";
  while (in.hasNextLine()) {
    String s = in.nextLine().trim();
    if (inp.equals("") && s.equals("")) {
      continue;
    }

    if (inp.equals("")) {
      inp = s;
    } else {
      addScenario(inp, s);
      inp = "";
    }
  }

  in.close();
}
```

3.3. Выводы по главе 3

1. Реализован алгоритм построения дерева сценариев работы. Программная реализация выполнялась на языке программирования *Java*. Приведена схема связей между используемыми классами.
2. Приведено описание базовых классов, используемых при реализации алгоритма построения дерева сценариев. Такими классами являются классы `StringActions`, `MyBooleanExpression`, `Transition`, `Node`, `StringScenario`. Проведен обзор полей и методов данных классов.
3. Приведено описание класса `ScenariosTree`, реализующего сущность «Дерево сценариев». Данный класс позволяет хранить и строить дерево сценариев работы по заданному множеству сценариев.

4. РАЗРАБОТКА АЛГОРИТМА ПОСТРОЕНИЯ ГРАФА СОВМЕСТИМОСТИ ВЕРШИН ДЕРЕВА СЦЕНАРИЕВ

В настоящем разделе приводится описание алгоритма построения графа совместимости вершин дерева сценариев. Сначала приводится описание рассматриваемого алгоритма, затем приводятся примеры работы разработанного алгоритма. Разработка алгоритма осуществлялась на основе методов динамического программирования.

4.1. Алгоритм построения графа совместимости вершин дерева сценариев

Для построения управляющего автомата необходимо «раскрасить» вершины дерева сценариев в заданное число цветов (равное числу состояний, которое задается в качестве одного из параметров алгоритма). При этом вершины одного цвета будут объединены в одно состояние результирующего автомата, а множество исходящих из состояния переходов будет строиться из объединения множеств ребер исходящих из вершин заданного цвета.

Для задания ограничений на раскраску построим так называемый *граф совместимости* вершин дерева сценариев. Множество вершин этого графа совпадает с множеством вершин дерева сценариев, поэтому в дальнейшем вершины графа и дерева различаться не будут. Ребра графа определяются следующим образом.

Вершины графа совместимости u и v соединены ребром (далее такие вершины будем называть *несовместимыми*), если существует последовательность пар $<e_1, values_1>\ldots<e_k, values_k>$ событий и наборов значений входных переменных, которая различает соответствующие вершины дерева. Будем говорить, что указанная последовательность *различает вершины* u и v, если выполняется совокупность следующих условий:

- из вершины u существует путь P_u, ребра которого помечены соответственно событиями $e_1\ldots e_k$ и такими охранными условиями $f_1\ldots f_k$, что наборы значений входных переменных $values_1\ldots values_k$ являются, соответственно, их выполняющими подстановками;
- аналогичный путь P_v существует из вершины v;
- для последних ребер путей P_u и P_v верно хотя бы одно из двух условий:
  пометки этих ребер различаются в части выходных воздействий;
  у охранных условий этих ребер есть общий выполняющий набор значений входных переменных, но они не совпадают как булевы функции.

Опишем алгоритм построения графа совместимости. Напомним, что множество вершин этого графа совпадает с множеством вершин дерева сценариев. Основной идеей данного алгоритма является метод динамического программирования [1, 5].

Для каждой вершины дерева сценариев v найдем все несовместимые с ней вершины. Обозначим $S(v)$ множество вершин, несовместимых с v. Будем вычислять значения функции $S(v)$, начиная с листьев дерева сценариев. Для каждого из листьев u множество $S(u)$ пусто по определению несовместимых вершин.

Покажем, как вычислить значение $S(v)$, если оно уже вычислено для всех значений «детей» вершины v. Переберем все вершины дерева – вершина u входит в множество $S(v)$, если существует пара ребер ux (помечено событием e, формулой f_1 и последовательностью действий A_1) и vy (помечено также событием e, формулой f_2 и последовательностью действий A_2) такая, что выполняется одно из трех:

- формулы f_1 и f_2 имеют общий выполняющий набор значений входных переменных, но не совпадают, как булевы функции. Тогда $<e, values>$, где как $values$ обозначена выполняющая подстановка f_1, – последовательность, различающая u и v;

- формулы f_1 и f_2 совпадают, как булевы функции, а последовательности A_1 и A_2 не совпадают. Тогда вершины u и v различает такая же последовательность;
- формулы f_1 и f_2 совпадают, как булевы функции, и вершина x входит в множество $S(y)$, посчитанное заранее. Тогда существует последовательность $<e_1, values_1>\ldots<e_k, values_k>$, различающая вершины x и y, а вершины v и u различает последовательность $<e, values><e_1, values_1>\ldots<e_k, values_k>$.

Время работы этого алгоритма составляет $O(n^2)$ (где n – число вершин в дереве сценариев), так как каждая пара ребер дерева сценариев в процессе работы алгоритма будет рассмотрена не более одного раза. При этом такое время работы достижимо, если заранее для каждой пары формул вычислено, равны ли они как булевы функции и имеют ли общий выполняющий набор значений входных переменных.

В худшем случае время работы этапа обработки формул составляет $O(2^{2m}n^2)$, где за m обозначено максимальное число входных переменных, использующихся в одном охранном условии. На практике число m не превышает четырех.

4.2. Примеры работы алгоритма построения графа совместимости

В настоящем разделе приведены примеры работы предложенного алгоритма построения графа совместимости. Сначала будет рассмотрен граф совместимости для небольшого дерева сценариев, затем будет рассмотрен граф совместимости дерева сценариев управления часами с будильником.

4.2.1. Построение графа совместимости небольшого дерева сценариев

На рис. 10 приведен граф совместимости для дерева сценариев, рассмотренного в разделе 2.2.1. Само дерево сценариев приведено на рис. 2.

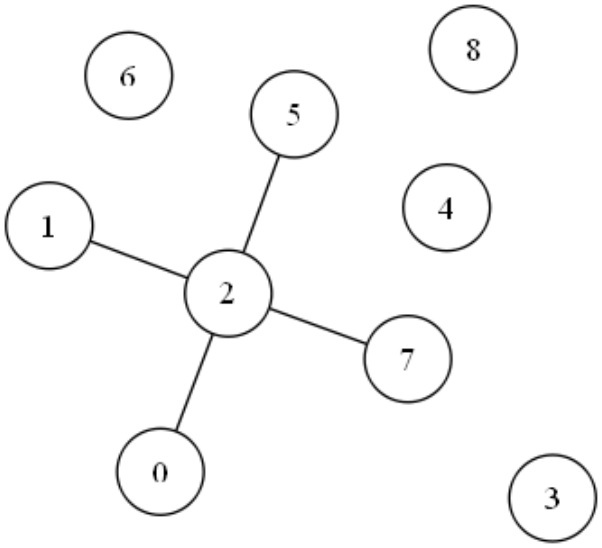

Рис. 10. Пример графа совместимости

4.2.2. Построение графа совместимости дерева сценариев управления часами с будильником

На рис. 11 приведен граф совместимости для дерева сценариев, рассмотренного в разделе 2.2.2. Само дерево сценариев приведено на рис. 5 – рис. 8. Полученный граф совместимости состоит из 116 вершин и 793 ребер.

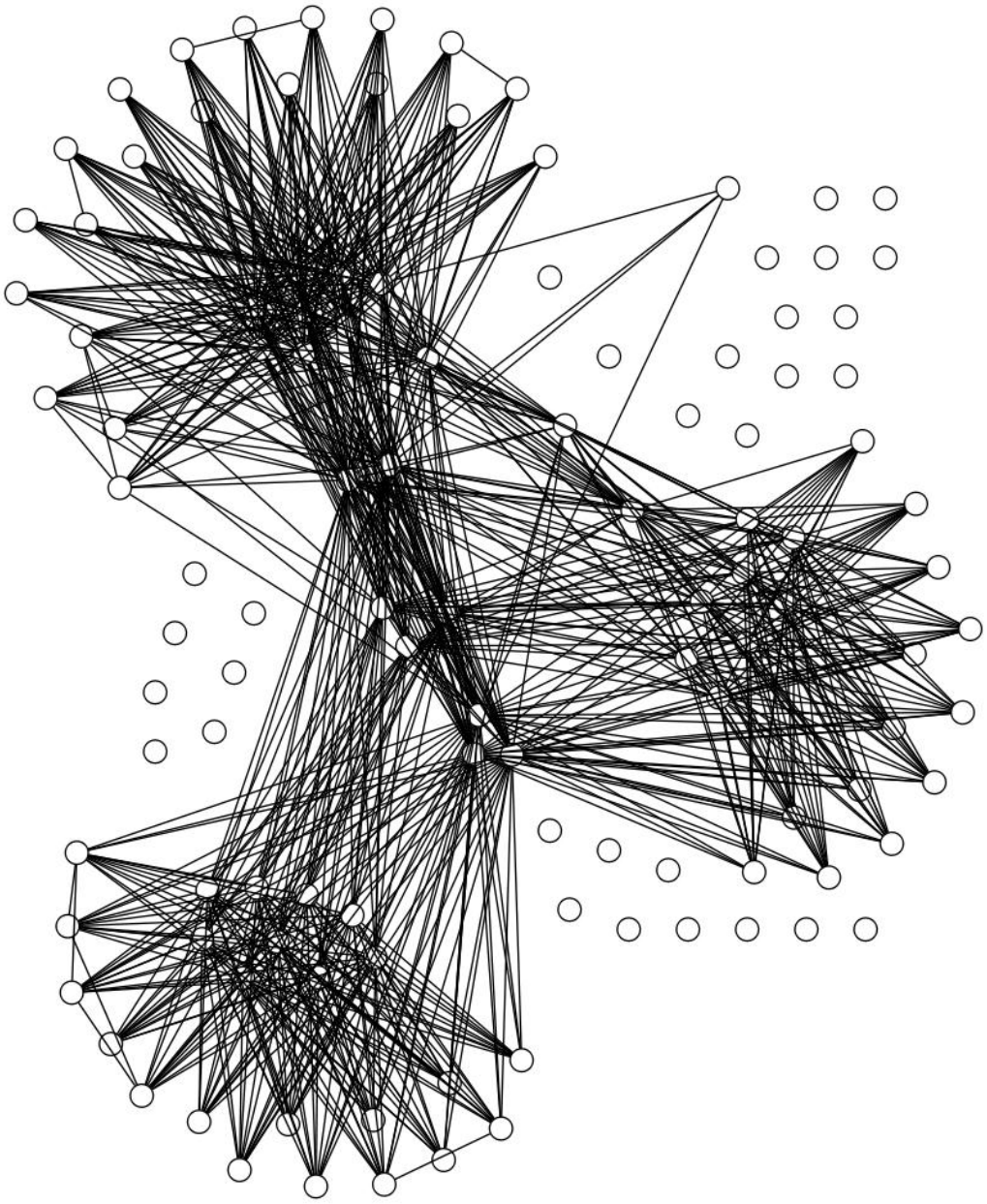

Рис. 11. Граф совместимости для дерева сценариев работы часов с будильником

4.3. Выводы по главе 4

1. Разработан алгоритм построения графа совместимости дерева сценариев. Разработка алгоритма осуществлялась на основе методов динамического программирования.
2. Приведен пример работы разработанного алгоритма на небольшом дереве сценариев, состоящем из девяти вершин. В полученном графе совместимости содержится четыре ребра – четыре пары несовместимых вершин.
3. Приведен пример работы разработанного алгоритма на дереве сценариев работы автомата управления часами с будильником. Данное дерево (и соответствующий граф совместимости) состоит из 116 вершин. В полученном графе совместимости содержится 793 ребра.

5. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ АЛГОРИТМА ПОСТРОЕНИЯ ГРАФА СОВМЕСТИМОСТИ ВЕРШИН ДЕРЕВА СЦЕНАРИЕВ

В настоящем разделе приведена реализация разработанного алгоритма построения графа совместимости.

5.1. Программная реализация алгоритма построения графа совместимости

Алгоритм построения графа совместимости реализован в классе `AdjacentCalculator`.

```
public class AdjacentCalculator
```

Статический метод `getAdjacent` принимает как параметр дерево сценариев `tree` типа `ScenariosTree`, описанного выше, и возвращает сопоставление типа `Map<Node, Set<Node>>` – каждой вершине сопоставляется набор вершин, несовместимых с ней. Именно в виде такого сопоставления хранится граф совместимости дерева сценариев.

```
public static Map<Node, Set<Node>> getAdjacent(ScenariosTree tree) {
  Map<Node, Set<Node>> ans = new HashMap<Node, Set<Node>>();
  calcNode(tree, tree.getRoot(), ans);
  return ans;
}
```

Метод `getAdjacent` вызывает рекурсивный метод `calcNode`. Методу параметрами передаются обрабатываемое дерево сценариев `tree`, текущая вершина данного дерева `node`, уже подсчитанная часть ответа `ans`.

Этот метод является «ленивой» реализацией метода динамического программирования – если ответ для переданной вершины дерева сценариев `node` уже был подсчитан, данный метод не станет выполнять вычисления заново.

```
private static void calcNode(ScenariosTree tree, Node
              node, Map<Node, Set<Node>> ans) {
  if (!ans.containsKey(node)) {
    HashSet<Node> adjacentSet = new HashSet<Node>();
    ans.put(node, adjacentSet);
    if (node.transitionsCount() == 0) {
      return;
    }

    for (Transition t1 : node.getTransitions()) {
      calcNode(tree, t1.getDst(), ans);
      for (Node other : tree.getNodes()) {
        if (other != node) {
          for (Transition t2 : other.getTransitions()) {
            if (t1.getEvent().equals(t2.getEvent())) {
              if (t1.getExpr() == t2.getExpr()) {
                if (!t1.getActions().equals(t2.getActions()) ||
                    ans.get(t1.getDst()).contains(t2.getDst())) {
```

```
                    adjacentSet.add(other);
                  }
                } else if (t1.getExpr().hasSolutionWith(t2.getExpr())) {
                  adjacentSet.add(other);
                }
              }
            }
          }
        }
      }
    }
```

5.2. Пример работы с полученным результатом

Приведем пример работы с экземпляром типа `Map<Node, Set<Node>>`, который возвращает метод `getAdjacent` класса `AdjacentCalculator`. Для этого выведем в файл `graph.gv` дерево сценариев в формате GraphViz [13]. Этот формат позволяет визуализировать граф одноименным программным средством. Пример визуализации графа совместимости данным средством приведен на рис. 11. Дерево сценариев при этом строится по набору сценариев, содержащихся в файле `scenarios`.

```
ScenariosTree tree = new ScenariosTree();
tree.load(("scenarios");

Map<Node, Set<Node>> adjacent = AdjacentCalculator.getAdjacent(tree);

PrintWriter pw = new PrintWriter(new File("graph.gv"));
pw.println("graph G {");
pw.println("    node [shape = circle, label=\"\", width=.20];");
for (Node node : tree.getNodes()) {
    pw.println(node.getNumber() + ";");
    for (Node other : adjacent.get(node)) {
        if (node.getNumber() < other.getNumber()) {
            pw.println("    " + node.getNumber() + " -- " +
                    other.getNumber() + ";");
        }
    }
}
pw.println("}");
pw.close();
```

5.3. Выводы по главе 5

1. Реализован алгоритм построения графа совместимости по дереву сценариев работы. Программная реализация выполнялась на языке программирования *Java*. Алгоритм построения графа совместимости реализован в классе `AdjacentCalculator`.

2. Приведен пример работы с результатом работы метода `getAdjacent` класса `AdjacentCalculator` – продемонстрирован способ вывода графа совместимости для дальнейшей визуализации с помощью программного средства GraphViz.

ЗАКЛЮЧЕНИЕ

В результате исследований на втором этапе работ по контракту были выполнены следующие работы:

- выполнена теоретическая оценка сложности задачи построения управляющего автомата по сценариям работы программы;
- разработан алгоритм построения дерева сценариев работы;
- выполнена программная реализация алгоритма построения дерева сценариев работы;
- разработан алгоритм построения графа совместимости вершин дерева сценариев;
- выполнена программная реализация алгоритма построения графа совместимости вершин дерева сценариев.

Результатом выполнения теоретической оценки сложности задачи являются доказательства принадлежности исследуемой задачи сложностным классам *NP*, *NP*-трудных задач, *NP*-полных задач. Приведены используемые в доказательстве понятия теории сложности и формализована поставленная задача.

Разработан и реализован на языке программирования *Java* алгоритм построения дерева сценариев. Разработка алгоритма осуществлялась с помощью методов теории графов. Приведены примеры работы данного алгоритма.

В результате разработки и реализации алгоритма построения графа совместимости было создано программное средство для построения графа совместимости. Разработка алгоритма осуществлялась на основе методов динамического программирования. Приведены примеры работы алгоритма.

Результаты выполненных работ позволяют утверждать, что научно-технический уровень исследований превышает уровень исследований в рассматриваемой области, проводимых в лучших исследовательских центрах мира.

СПИСОК ЛИТЕРАТУРЫ

1. *Кларк Э., Грамберг О., Пелед Д.* Верификация моделей программ. М.: МЦНМО, 2002.
2. *Кормен Т.*, *Лейзерсон Ч.*, *Ривест Р.*, *Штайн К.* Алгоритмы. Построение и анализ. М.: Вильямс, 2010.
3. *Николенко С. И., Тулупьев А. Л.* Самообучающиеся системы. М.: МЦНМО, 2009.
4. *Поликарпова Н. И., Шалыто А. А.* Автоматное программирования. СПб: Питер, 2009.
5. *Скиена С.* Алгоритмы. Руководство по разработке. СПб: БХВ-Петербург, 2011.
6. *Стратонович Р.* Теория информации. М.: Сов. радио, 1975.
7. *Хопкрофт Дж., Мотвани Р., Ульман Дж.* Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2002.
8. *Biere A.* Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Technical Report 10/1, August 2010. FMV Reports Series, Institute for Formal Models and Verification. Johannes Kepler University. Altenbergerstr. 69. 4040 Linz, Austria.
9. *Bouchard C.* Boolean Expression Solver. http://sourceforge.net/projects/bool-expr-solve/.
10. *CryptoMiniSat SAT-solver*. http://www.msoos.org/cryptominisat2.
11. *Een N.*, *Sörensson N.* An extensible SAT-solver. Chalmers University of Technology, Sweden, 2003. http://minisat.se/downloads/MiniSat.pdf.
12. *Gold E. M.* Complexity of automaton identification from given data // Information and Control. 1978. № 37, pp. 302−320.
13. *Graphviz* – Graph Visualization Software. http://www.graphviz.org/.
14. *Heule M.*, *Verwer S.* Exact DFA Identification Using SAT Solvers // Grammatical Inference: Theoretical Results and Applications 10th International Colloquium (ICGI 2010), pp. 66-79. Lecture Notes in Computer Science. 2010. V. 6339, .
15. zChaff SAT-solver. http://www.princeton.edu/~chaff/zchaff.html.

ПРИЛОЖЕНИЕ 1. STRINGACTIONS.JAVA

```
package actions;

import java.util.Arrays;

public class StringActions {
  private String[] actions;

  public StringActions(String str) {
    str = str.trim();
    if (str.equals("")) {
      this.actions = new String[0];
    } else {
      this.actions = str.split(",");
    }
    for (int i = 0; i < actions.length; i++) {
      this.actions[i] = this.actions[i].trim();
    }
  }

  public String[] getActions() {
    return actions;
  }

  public int size() {
    return actions.length;
  }

  public boolean equals(StringActions other) {
    return Arrays.deepEquals(this.actions, other.actions);
  }

  public String toString() {
    String res = "";
    for (int i = 0; i < actions.length; i++) {
      if (i > 0) {
        res += ", ";
      }
      res += actions[i];
    }
    return res;
  }
}
```

ПРИЛОЖЕНИЕ 2. MYBOOLEANEXPRESSION.JAVA

```
package bool;

import java.text.ParseException;
import java.util.Arrays;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.regex.Pattern;

public class MyBooleanExpression {
  private String repr;
  private String[] variables;
  private Map<Map<String, Boolean>, Boolean> truthTable;

  public MyBooleanExpression(String expression) throws ParseException {
    expression = expression.replaceAll(" ", "").replaceAll("!", "~");
    this.repr = expression;
    Pattern pattern = Pattern.compile("[()~&|=>+]");
    String[] vars = pattern.split(expression);

    HashSet<String> varsSet = new HashSet<String>(Arrays.asList(vars));
    varsSet.removeAll(Arrays.asList(new String[]{"", "1", "0"}));
    this.variables = varsSet.toArray(new String[0]);

    String shortExpr = new String(expression);
    for (int i = 0; i < variables.length; i++) {
      shortExpr = shortExpr.replaceAll(variables[i], "" +
                (char)('a' + i));
    }

    BooleanExpression booleanExpression = new
                BooleanExpression(shortExpr);
    Map<Map<String, Boolean>, Map<BooleanExpression, Boolean>> truthTable
                 = new TruthTable(booleanExpression).getResults();
    this.truthTable = new HashMap<Map<String,Boolean>, Boolean>();

    for (Map<String, Boolean> map : truthTable.keySet()) {
      Map<BooleanExpression, Boolean> booleanMap =
                truthTable.get(map);
      boolean val = booleanMap.containsValue(true);
      this.truthTable.put(map, val);
    }
  }

  public boolean hasSolution() {
    return truthTable.containsValue(true);
```

```
  }

  public boolean isTautology() {
    return !truthTable.containsValue(false);
  }

  public boolean equals(MyBooleanExpression other) {
    if (other.repr.equals(repr)) {
      return true;
    }

    MyBooleanExpression e = null;
    try {
      e = new MyBooleanExpression("(" + repr + ")=(" + other.repr + ")");
    } catch (ParseException ex) {
      ex.printStackTrace();
      System.exit(1);
    }
    return e.isTautology();
  }

  private Map<MyBooleanExpression, Boolean> hasSolutionWithRes;

  public boolean hasSolutionWith(MyBooleanExpression other) {
    if (hasSolutionWithRes == null) {
      hasSolutionWithRes = new HashMap<MyBooleanExpression, Boolean>();
    }

    if (hasSolutionWithRes.containsKey(other)) {
      return hasSolutionWithRes.get(other);
    }

    MyBooleanExpression e = null;
    try {
      e = new MyBooleanExpression("(" + repr + ")&(" + other.repr + ")");
    } catch (ParseException ex) {
      ex.printStackTrace();
      System.exit(1);
    }
    boolean res = e.hasSolution();
    hasSolutionWithRes.put(other, res);
    return res;
  }

  public String toString() {
    return repr;
  }
}
```

ПРИЛОЖЕНИЕ 3. TRANSITION.JAVA

```
package stuctures;

import actions.StringActions;
import bool.MyBooleanExpression;

public class Transition {
  private String event;
  private MyBooleanExpression expr;
  private StringActions actions;
  private Node src;
  private Node dst;

  public Transition(Node src, Node dst, String event,
                MyBooleanExpression expr, StringActions actions) {
    this.src = src;
    this.dst = dst;
    this.event = event;
    this.expr = expr;
    this.actions = actions;
  }

  public String getEvent() {
    return event;
  }
  public Node getSrc() {
    return src;
  }
  public Node getDst() {
    return dst;
  }
  public StringActions getActions() {
    return actions;
  }
  public MyBooleanExpression getExpr() {
    return expr;
  }

  public String toString() {
    String res = src.getNumber() + " -> " + dst.getNumber();
    res += "  " + event + " [" + expr.toString() + "] " +
                actions.toString();
    return res;
  }
}
```

ПРИЛОЖЕНИЕ 4. NODE.JAVA

```
package stuctures;

import java.util.*;
import actions.StringActions;
import bool.MyBooleanExpression;

public class Node {
  private int number;
  private Map<String, Transition> transitions;

  public Node(int number) {
    this.number = number;
    transitions = new HashMap<String, Transition>();
  }

  public int getNumber() {
    return number;
  }

  public boolean hasTransition(String event, MyBooleanExpression expr) {
    return transitions.containsKey(event + "[" + expr.toString() + "]");
  }

  public void addTransition(String event, MyBooleanExpression expr,
                            StringActions actions, Node dst) {
    Transition transition = new Transition(this, dst, event,
                expr, actions);
    transitions.put(event + "[" + expr.toString() + "]", transition);
  }
  public Transition getTransition(String event,
                MyBooleanExpression expr) {
    return transitions.get(event + "[" + expr.toString() + "]");
  }
  public Collection<Transition> getTransitions() {
    return transitions.values();
  }

  public Node getDst(String event, MyBooleanExpression expr) {
    return transitions.get(event + "[" + expr.toString() + "]").getDst();
  }

  public int transitionsCount() {
    return transitions.size();
  }
}
```

ПРИЛОЖЕНИЕ 5. STRINGSCENARIO.JAVA

```
package scenario;

import java.text.ParseException;
import java.util.ArrayList;
import java.util.List;

import bool.MyBooleanExpression;
import actions.StringActions;

public class StringScenario {
  private static List<MyBooleanExpression> exprList = new
               ArrayList<MyBooleanExpression>();

  private String[] events;

  private MyBooleanExpression[] expressions;

  private StringActions[] actions;

  public StringScenario(String input, String output) throws
               ParseException {
    String[] events = input.split(";");
    String[] actions = (output + " ").split(";");
    if (actions.length != events.length) {
      throw new ParseException("events length " +
               events.length + " != actions length " +
               actions.length, 0);
    }

    int n = actions.length;

    this.events = new String[n];
    this.expressions = new MyBooleanExpression[n];
    this.actions = new StringActions[n];

    for (int i = 0; i < n; i++) {
      MyBooleanExpression newExpr;

      if (events[i].contains("[")) {
        String[] p = events[i].split("\\[");
        this.events[i] = p[0].trim();
        newExpr = new MyBooleanExpression(p[1].replace(']', ' '));
      } else {
        this.events[i] = events[i].trim();
        newExpr = new MyBooleanExpression("1");
      }
```

```
      MyBooleanExpression eq = null;
      for (MyBooleanExpression e2 : exprList) {
        if (newExpr.equals(e2)) {
          eq = e2;
          break;
        }
      }
      if (eq == null) {
        exprList.add(newExpr);
        eq = newExpr;
      }
      this.expressions[i] = eq;
      this.actions[i] = new StringActions(actions[i]);
    }
  }

  public int size() {
    return events.length;
  }

  public String getEvent(int pos) {
    return events[pos];
  }

  public MyBooleanExpression getExpr(int pos) {
    return expressions[pos];
  }

  public StringActions getActions(int pos) {
    return actions[pos];
  }

  public String toString() {
    String inp = "";
    String out = "";
    for (int i = 0; i < size(); i++) {
      if (i > 0) {
        inp += "; ";
        out += "; ";
      }
      inp += events[i] + "[" + expressions[i].toString() + "]";
      out += actions[i].toString();
    }
    return inp + "\n" + out;
  }
}
```

ПРИЛОЖЕНИЕ 6. SCENARIOSTREE.JAVA

```
package stuctures;

import java.io.*;
import java.util.*;
import java.text.ParseException;

import actions.StringActions;
import bool.MyBooleanExpression;

import scenario.StringScenario;

public class ScenariosTree {
  private Node root;
  private List<Node> nodes;

  public ScenariosTree() {
    this.root = new Node(0);
    this.nodes = new ArrayList<Node>();
    this.nodes.add(root);
  }

  public Node getRoot() {
    return root;
  }

  public List<Node> getNodes() {
    return nodes;
  }

  public int nodesCount() {
    return nodes.size();
  }

  public void load(String filepath) throws
                FileNotFoundException, ParseException {
    Scanner in = new Scanner(new File(filepath));

    String inp = "";
    while (in.hasNextLine()) {
      String s = in.nextLine().trim();
      if (inp.equals("") && s.equals("")) {
        continue;
      }

      if (inp.equals("")) {
        inp = s;
```

```
    } else {
      addScenario(inp, s);
      inp = "";
    }
  }

  in.close();
}

private void addScenario(String input, String output)
              throws ParseException {
  StringScenario scenario = new StringScenario(input, output);

  Node node = root;
  for (int i = 0; i < scenario.size(); i++) {
    addTransition(node, scenario.getEvent(i),
              scenario.getExpr(i),
              scenario.getActions(i));
    node = node.getDst(scenario.getEvent(i),
              scenario.getExpr(i));
  }
}

private void addTransition(Node src, String event,
              MyBooleanExpression expr, StringActions
              actions)
    throws ParseException {
  if (src.hasTransition(event, expr)) {
    Transition t = src.getTransition(event, expr);
    if (!t.getActions().equals(actions)) {
      throw new ParseException("bad transition add in node
              " + src.getNumber() + ": " + t.getActions()
          + " != " + actions, 0);
    }
  } else {
    Node dst = new Node(nodes.size());
    nodes.add(dst);
    src.addTransition(event, expr, actions, dst);
  }
}

public String toString() {
  final String[] colors = new String[] { "blue", "red",
              "green", "black", "yellow", "cyan", "gold",
              "green4", "navy", "orange", "brown", "crimson" };

  String res = "# generated file, don't try to modify\n";
  res += "# command: dot -Tpng <filename> > tree.png\n";
```

```
    res += "digraph ScenariosTree {\n";

    for (Node node : nodes) {
      for (Transition t : node.getTransitions()) {
        res += "    " + t.getSrc().getNumber() + " -> " +
                t.getDst().getNumber();
        res += " [label = \"" + t.getEvent() + " [" +
                t.getExpr().toString() + "] (" +
                 t.getActions().toString() + ") \"];\n";
      }
    }

    res += "}";
    return res;
  }
}
```