Министерство образования и науки Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

Факультет **Информационных технологий и программирования**

Направление **Прикладная математика и информатика**

Академическая степень **Магистр прикладной математики и информатики**

Кафедра **Компьютерные технологии** Группа **6538**

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

Построение управляющих конечных автоматов по сценариям работы на основе решения задачи удовлетворения ограничений

Автор магистерской диссертации **В.И. Ульянцев**

Научный руководитель **А.А. Шалыто**

К защите допустить
Зав. кафедрой **В.Н. Васильев**

«__» _____ 2013 г.

Санкт-Петербург, 2013

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В последние годы для решения разнообразных задач все чаще применяется автоматное программирование [9]. В рамках этого подхода поведение программ описывается с помощью детерминированных конечных автоматов. Важным достоинством автоматных программ является возможность их автоматической верификации [5], что является существенным свойством для систем с повышенными требованиями к надежности.

Для этих систем важную роль играет влияние человеческого фактора на процесс разработки. В связи с этим актуальной задачей является разработка методов автоматизации построения конечных автоматов. Такие методы позволят снизить влияние человеческого фактора на процесс разработки.

Одним из способов автоматизированного построения конечных автоматов является их построение по экспертным данным [8], которым искомый автомат должен соответствовать. Альтернативным методом автоматизированного построения автоматов является применение эволюционных алгоритмов [2, 16, 29], которое подразумевает введение функции приспособленности, определенной над множеством автоматов.

В известных работах, в которых предлагаются методы построения автоматов по экспертным данным, к искомой модели предъявляется требование непротиворечивости – не должно быть двух переходов, исходящих из одного состояния управляющего автомата и одновременно выполнимых при некоторой комбинации события и входных переменных. В некоторых системах к автоматам управления предъявляется требование полноты – любой комбинации события и входных переменных должен соответствовать переход в каждом состоянии. Однако существующие методы построения автоматов не предусматривают возможности построения моделей, удовлетворяющих требованию полноты.

В настоящей диссертации проводится разработка, реализация и экспериментальное исследование метода построения управляющих автоматов, удовлетворяющих не только требованию непротиворечивости, но и требованию полноты. Метод основан на решении задачи удовлетворения ограничений (*constraint satisfaction problem, CSP*). Пусть имеется набор переменных x_1, …, x_n, для каждой из которых задано множество допустимых значений. Задача удовлетворения ограничений заключается в подборе таких значений x_1, …, x_n, что выполняются все заданные ограничения на эти переменные. Примерами ограничений являются такие утверждения, как $2x_1 + x_2 = x_3$, $(x_1 = x_2) \Rightarrow (x_3 = 1)$.

Метод построения управляющих автоматов в качестве входных экспертных данных использует безошибочные сценарии работы программы. Метод показывает существенно более высокую производительность по сравнению с генетическими и эволюционными алгоритмами, которые традиционно применяются для решения указанной задачи. Производительность метода позволяет строить управляющие автоматы с десятками состояний, а размер входных данных сможет составлять тысячи событий и выходных воздействий. В качественном отношении метод позволяет строить автоматные программы, удовлетворяющие не только требованию непротиворечивости, но и требованию полноты, что не позволяет делать ни один из существующих в настоящее время методов машинного обучения.

Исследование поддержано федеральной целевой программой «Научные и научно-педагогические кадры инновационной России» на 2009–2013 годы. Мероприятие 1.2.1. «Проведение научных исследований научными группами под руководством докторов наук». Направление «Информатика». Тема «Разработка методов построения управляющих конечных автоматов по обучающим примерам на основе решения задачи удовлетворения ограничений».

1. АНАЛИТИЧЕСКИЙ ОБЗОР

В настоящем разделе приводятся результаты аналитического обзора. Аналитический обзор проводился по следующим направлениям:

- методы построения конечных автоматов по обучающим примерам;
- программные средства для решения задачи удовлетворения ограничений.

1.1. МЕТОДЫ ПОСТРОЕНИЯ КОНЕЧНЫХ АВТОМАТОВ ПО ОБУЧАЮЩИМ ПРИМЕРАМ

В настоящем подразделе приводится обзор существующих методов построения конечных автоматов по обучающим примерам.

1.1.1. Типы автоматных моделей

В данном пункте приводится описание основных типов автоматных моделей, в соответствии с определениями, приведенными в [9]. Абстрактные конечные автоматы принято описывать в следующих терминах. Задано конечное множество символов *X*, которое называется (входным) алфавитом. Множество всех возможных строк (последовательностей, слов), составленных из символов алфавита *X* обозначается X^*. Пустая последовательность символов обозначается ε, $\varepsilon \in X^*$. Подмножество *L* множества всех цепочек над алфавитом *X*, $L \subset X^*$, называется *языком*. Рассматривается следующая проблема: задан язык $L \subset X^*$ и строка $\xi \in X^*$. Необходимо определить, принадлежит ли указанное слово языку ($\xi \in L$).

Если абстрактный вычислитель способен решить эту проблему для определенного языка $L \subset X^*$ и произвольной строки $\xi \in X^*$, то говорят, что вычислитель распознает язык *L*. Таким образом, абстрактные автоматы описываются в терминах тех языков, которые они распознают. Различные автоматные модели могут распознавать разные классы языков или, другими словами, обладают разной вычислительной

мощностью (вычислительная мощность модели абстрактных автоматов тем больше, чем шире класс распознаваемых ими языков).

Детерминированный конечный автомат (*ДКА*) или конечный автомат-распознаватель – это пятерка $\langle X, Y, \delta, y_0, F \rangle$, где X – конечный алфавит входных символов, Y – конечное множество состояний, $\delta : X \times Y \to Y$ – функция переходов, $y_0 \in Y$ – начальное (стартовое) состояние, $F \subset Y$ – множество допускающих состояний [9].

Расширенная функция переходов $\hat{\delta} : X^* \times Y \to Y$, сопоставляющая новое состояние текущему состоянию и слову, определяется индуктивно следующим образом [13]:

$$\forall y \in Y : \hat{\delta}(\varepsilon, y) = y;$$

$$\forall y \in Y \ \forall \xi \in X^* \ \forall x \in X : \hat{\delta}(\xi x, y) = \delta(x, \hat{\delta}(\xi, y)).$$

В таком случае, если $\hat{\delta}(\xi, y_0) \in F$ (стартуя в начальном состоянии и обработав строку ξ, автомат оказывается в одном из допускающих состояний), говорят, что он *допускает* эту цепочку. Множество допускаемых цепочек образует язык *L*, распознаваемый ДКА: $L = \{\xi \in X^* \mid \hat{\delta}(\xi, y_0) \in F\}$. Класс языков, распознаваемых ДКА, называют *регулярными* языками. Известно, что он совпадает с классом языков, описываемых регулярными выражениями и автоматными грамматиками [13].

Для того чтобы наделить модель абстрактного конечного автомата способностью не только давать ответ типа «да/нет», но и выполнять какие-то преобразования, в модель добавляют конечный алфавит выходных символов *Z* и *функцию выхода* φ. Если функция выхода имеет вид $\varphi : X \times Y \to Z$ (переходы помечены выходными символами), то вычислитель называется *автоматом Мили*, а если $\varphi : Y \to Z$ (выходными символами помечены состояния) – *автоматом Мура*. Таким образом, рассматривается шестерка $\langle X, Y, Z, \delta, \varphi, y_0 \rangle$. Она определяет автоматное отображение $f : X^* \to Z^*$ (преобразование, выполняемое автоматом) следующим образом:

$$f(\varepsilon) = \varepsilon \text{ ;}$$

$$\forall \xi \in X^* \; \forall x \in X : f(\xi x) = f(\xi)\varphi(x, \hat{\delta}(\xi, y_0)) \text{ .}$$

Автоматные отображения [9] – это отображения «без предсказания»: перерабатывая слово слева направо, они «не заглядывают вперед». Например, отображение, которое сопоставляет цепочке ее саму, записанную в обратном порядке, не является автоматным.

Понятие *управляющего автомата Мура* (*Moore machine*) аналогично понятию абстрактного автомата Мура, введенному выше. В таком автомате выходное воздействие зависит только от состояния и не зависит от входного воздействия. На рис. 1 приведен пример графа переходов автомата Мура. Приведенный автомат содержит четыре состояния (на рисунке не пронумерованы), алфавит содержит два символа $X = \{x, \neg x\}$, алфавит выходных воздействий содержит три символа $Z = \{z_1, z_2, z_3\}$.

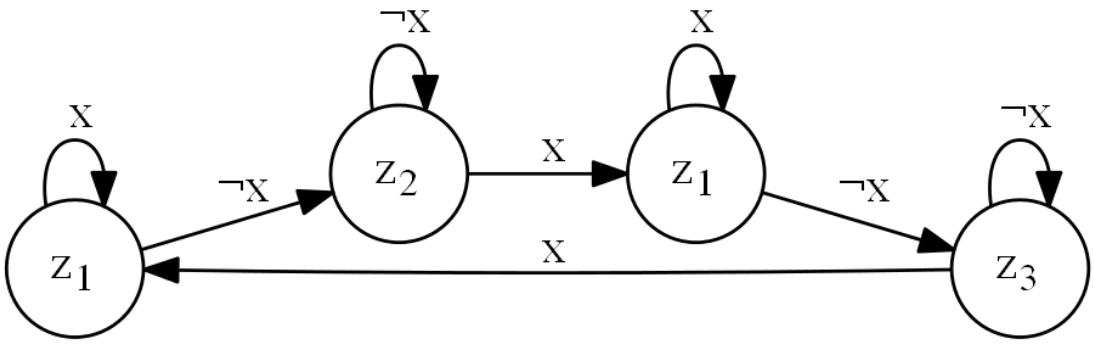

Рис. 1. Пример автомата Мура

По аналогии с абстрактными автоматами, структурный автомат, выходные воздействия которого зависят не только от состояния, но и от входных воздействий, называется *автоматом Мили* (*Mealy machine*) [9]. Известно [12], что для любого автомата Мили можно построить эквивалентный ему автомат Мура. Число состояний в таком автомате будет не меньше, чем в исходном.

На рис. 2 приведен пример автомата Мили, поведение которого совпадает с поведением автомата Мура, приведенного на рис. 1. На рисунке состояния

пронумерованы, переходы помечены входным и выходным символами, записанными через «/».

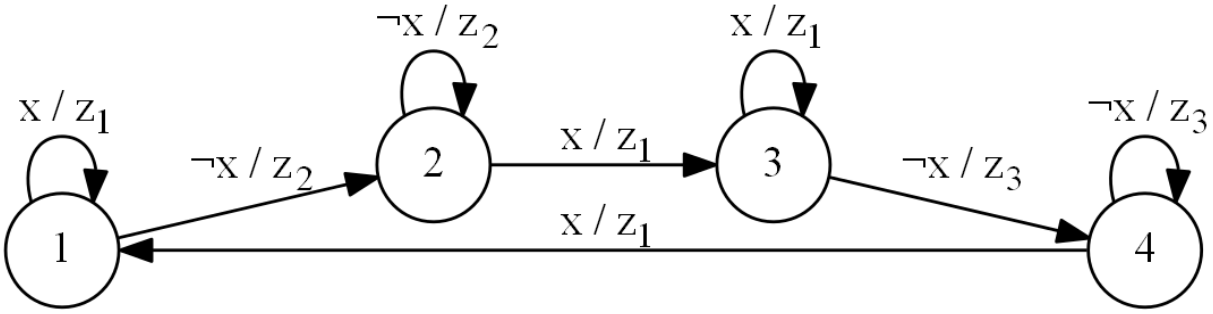

Рис. 2. Пример автомата Мили

Если часть выходных переменных автомата зависит только от состояний, а остальные – также и от входных воздействий, удобно разделить функцию выходов φ на две составляющие: $\varphi_1(y)$ и $\varphi_2(x, y)$. В структурную схему автомата в этом случае вводится не один, как в автоматах Мура и Мили, а два выходных преобразователя. Такие автоматы называются смешанными, автоматами Мура-Мили или С-автоматами [9].

В дальнейшем будет введено понятие управляющего конечного автомата (в англоязычных работах чаще всего называется *Extended Finite State Machine*, *EFSM*), метод построения которых описывается в настоящей диссертации.

1.1.2. Построение конечных автоматов-распознавателей по обучающим примерам

Из теории формальных языков известно, что конечные детерминированные автоматы способны распознавать регулярные языки [13]. В связи с этим актуальна задача построения автомата, распознающего по множеству примеров некий язык. Задача может быть усилена до построения автомата с минимальным количеством состояний. В 1978 г. Голдом было показано [26], что эта задача является *NP*-трудной. Тем не менее, при некоторых ограничениях на входные данные задача может быть

решена за полиномиальное время. В других случаях могут либо быть использованы различные эвристики, работающие за полиномиальное время, но, тем не менее, не дающие гарантий минимальности создаваемого автомата, либо эвристики, в общем случае работающие за экспоненциальное время.

Входными данными для задачи построения конечного автомата-распознавателя по обучающим примерам являются два множества слов S^+ и S^-. Слова из первого множества должны допускаться автоматом, слова из второго – нет. Такое свойство автомата далее будем называть *совместимостью* с S^+ и S^-.

Методы построения автоматов по обучающим примерам иногда называют методами *машинного обучения*. В соответствии с книгой [7] алгоритм является алгоритмом машинного обучения, если он улучшает свое поведение по мере накопления опыта. Это означает, что алгоритм обучает параметры модели либо на заранее подготовленных тестовых примерах (верно для рассматриваемых задач построения автоматов), либо на собственных ошибках, и со временем решает поставленную задачу все лучше и лучше. Некоторые алгоритмы машинного обучения способны замечать ранее неизвестные закономерности в данных, выделять знания, которых раньше не было.

Модели, основанные на состояниях, широко применяются в машинном обучении. Примером таких моделей являются скрытые Марковские модели. 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.

Далее приводится обзор алгоритмов машинного обучения для задачи построения автоматов-распознавателей по обучающим примерам.

1.1.2.1. Алгоритмы, основанные на объединении состояний

Основной идеей работы алгоритмов, основанных на объединении (слиянии) состояний является построение *префиксного дерева* (*APTA, augmented prefix tree acceptor*), и последовательное отождествление его состояний, в результате чего получается конечный автомат.

Префиксное дерево является частным случаем автомата-распознавателя. Состояния префиксного дерева могут быть *помеченными*, то есть допускающими или недопускающими, или *непомеченными*. Для каждого слова из набора тестов в соответствующем ему состоянии дерева присутствует пометка о допуске или недопуске. Пример префиксного дерева для $S^+ = \{a, abaa, bb\}$, $S^- = \{b, abb\}$ приведен на рис. 3.

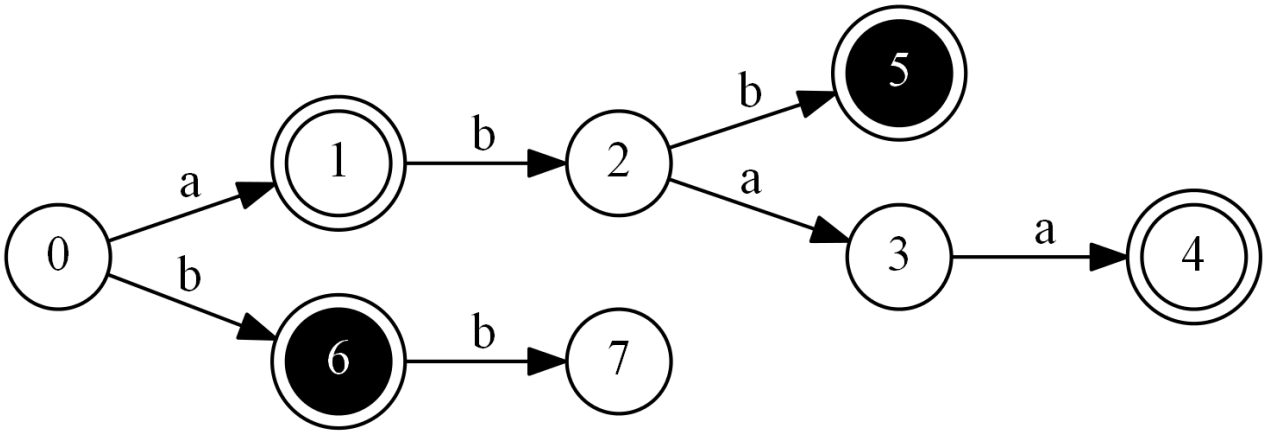

Рис. 3. Пример префиксного дерева

Хронологически первым алгоритмом для решения задачи построения минимального автомата, совместимого с входным набором тестов, который работает за полиномиальное время, стал алгоритм [45], предложенный Трахтенбротом (Trakhtenbrot) и Барздином (Barzdin) в 1973 г. Для краткости будем называть его *TB*-алгоритмом. Необходимым условием для работы *TB*-алгоритма является *полнота* набора тестов: в S^+ и S^- должны содержаться все слова длиной до некоторого натурального числа. Алгоритм строит минимальный автомат, совместимый с S^+ и S^-.

TB-алгоритм начинает свою работу с построения префиксного дерева. Поскольку входной набор тестов полон, все состояния дерева будут помеченными. Алгоритм представляет собой два вложенных цикла, перебирающих состояния дерева в порядке обхода в ширину. Каждая пара различных состояний u и v (пусть для определенности u дальше от корня, чем v) проверяется на эквивалентность: переходы из обоих состояний по всем строкам длины k, где k – минимум из высот поддеревьев с корнями в u и v, должны приводить в состояния с одинаковыми пометками. Если состояния эквивалентны, то поддерево с корнем в u удаляется, а ссылка родителя u переставляется на v. В противном случае слияние состояний невозможно. Алгоритм работает за время $O(ph^2)$, где p – размер префиксного дерева, h – число состояний в минимальном автомате.

Лангом [32] была предложена улучшенная версия *TB*-алгоритма, которая получила название *Traxbar*. Этот алгоритм не требует полноты набора тестов. Пары состояний для слияния перебираются в порядке обхода в ширину, как и в *TB*-алгоритме, однако само слияние состояний производится более сложным способом. Для проверки двух состояний на совместимость делается попытка осуществить их слияние, сохранив информацию, с помощью которой можно вернуться назад. При слиянии состояний u и v, где u дальше от корня в порядке обхода в ширину, метки из поддерева с корнем в u должны быть скопированы в подграф, получившийся из поддерева с корнем в v. Алгоритм *Traxbar* не дает гарантий минимальности результирующего автомата, однако порядок обхода состояний для слияния повышает вероятность того, что первые слияния, которые осуществляет алгоритм, ведут к построению минимального автомата.

Алгоритм Голда [26] – еще один алгоритм построения минимального автомата по тестам, работающий за полиномиальное время. Для его работы не требуется полноты набора тестов, однако набор должен содержать некоторое особое

характеристическое множество регулярного языка, иначе алгоритм построит несовместимый с S^+ или S^- автомат.

Алгоритм, названный *RPNI* (*regular positive and negative inference*) [40], был предложен в 1992 г. В начале алгоритма строится префиксное дерево по словарю S^+ (оно не будет допускать слова из S^-). После этого производится упорядоченный поиск среди разбиений множества вершин префиксного дерева на классы эквивалентности: в двойном цикле, перебирающем пары различных вершин (u, v) префиксного дерева, осуществляется попытка слить классы эквивалентности состояний, к которым принадлежат u и v. В результате слияния может получиться недетерминированный автомат, для устранения этого проводятся дополнительные слияния. В случае если получившийся детерминированный автомат не допускает слова из S^-, этот автомат и соответствующее ему разбиение заменяют текущие. В противном случае, текущий автомат и текущее разбиение не меняются. Алгоритм работает за время $O((m + n)\, n^2)$, где n – суммарная длина слов из S^+, m – суммарная длина слов из S^-.

В 1998 г. было проведено соревнование *Abbadingo One* [31], целью которого являлось развитие алгоритмов построения автоматов-распознавателей по обучающим примерам. Каждая задача соревнования представляла собой два набора тестов. Первый набор являлся входным для алгоритма (S^+ и S^-) и генерировался по случайному автомату A заданного размера. Другой набор тестов также генерировался по автомату A, но использовался для сравнения A с автоматом, полученным алгоритмом участника соревнования. Задача считалась решенной, если 99 % слов из второго набора тестов верно распознавалась алгоритмом участника.

Одним из двух победителей соревнования стал алгоритм *объединения состояний на основе свидетельств*, или *EDSM* (*evidence-driven state merging*). Данный алгоритм начинает свою работу с построения префиксного дерева по словарям S^+ и S^-. Далее на каждом шаге алгоритма происходит слияние некоторых состояний текущего автомата. Отличительной особенностью алгоритма является то, что для всех возможных на

текущем шаге слияний производится оценка, основанная на входных данных, и осуществляется слияние с наилучшей оценкой. Алгоритм завершается при отсутствии возможных слияний.

Модификацией алгоритма *EDSM* является алгоритм *Blue-Fringe*, предложенный после подведения итогов соревнования *Abbadingo One*. Данный алгоритм накладывает дополнительные ограничения на порядок слияний вершин префиксного дерева.

На каждом шаге алгоритма состояния автомата разделены на три множества. Каждому множеству условно соответствует цвет. Множеству необработанных состояний соответствует белый цвет. Множество красных состояний – попарно не сливаемые состояния, которые являются частью результирующего автомата. Все состояния, не являющиеся красными, в которые ведут переходы из красных состояний, являются синими. Пример раскраски состояний после нескольких шагов алгоритма приведен на рис. 4.

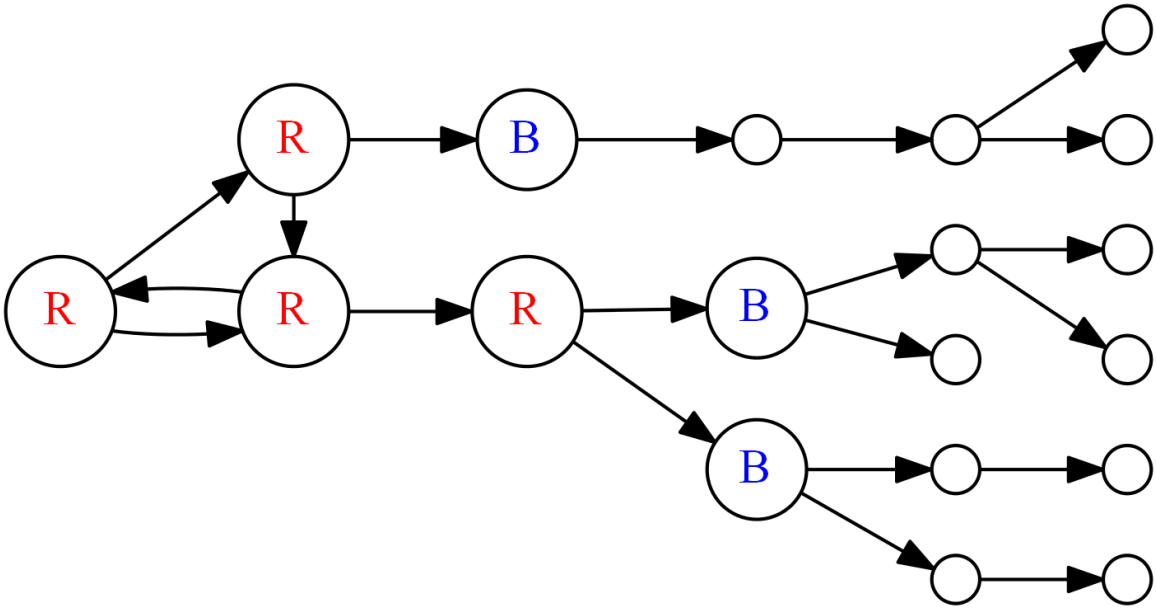

Рис. 4. Пример автомата после нескольких шагов алгоритма *Blue-Fringe*

На каждом шаге алгоритма все синие состояния, которые нельзя слить ни с одним из красных, перекрашиваются в красный цвет. Затем выбирается пара состояний красного и синего цвета с наибольшим значением функции, оценивающей слияния, и

производится слияние выбранной пары состояний. В процессе работы алгоритма производится перекраска в синий цвет белых состояний, которые стали потомками красных.

В [22] приводится сравнение алгоритмов *Blue-Fringe* и *EDSM*. Первый алгоритм незначительно уступил второму по числу решенных задач, но среди задач, использовавшихся в сравнении, было много и тех, на которых лучше оказался *Blue-Fringe*, и тех, на которых лидировал *EDSM*. Время работы *Blue-Fringe* было оценено сверху как $O(ph^3)$, где p – размер префиксного дерева, h – число состояний в результирующем автомате. Для реализации *EDSM*, использовавшейся в сравнении, точная верхняя оценка не была установлена.

Другой модификацией алгоритма *EDSM* является *W-EDSM* (*Windowed-EDSM*). В предлагаемом подходе рассматриваются только слияния состояний, находящихся в некотором «окне». «Окно» представляет собой множество состояний, находящихся в пределах некоторого числа шагов работы обхода в ширину, запущенного от корня дерева. Если в результате слияния размер «окна» уменьшается, в него добавляются новые вершины, также в порядке обхода в ширину. В случае отсутствия возможных слияний внутри «окна» его размер удваивается. Как и в классической версии *EDSM*, алгоритм завершается, когда не остается допустимых слияний.

За счет уменьшения числа рассматриваемых слияний время работы *W-EDSM* меньше, чем у обычного *EDSM*. Это означает, что *W-EDSM* может быть применен к задачам с большей размерностью. Тем не менее, уменьшение времени работы может компенсироваться ухудшением качества результирующего автомата вследствие более высокой вероятности совершить неправильное слияние.

В работах [17, 33, 34, 35, 39, 44] приведены эволюционные алгоритмы для построения автоматных моделей по экспертным данным, в некоторых случаях их сравнение с алгоритмами, основанными на объединении состояний. Многие из данных алгоритмов подробно описаны в [14].

1.1.2.2. Алгоритмы, основанные на сведении к другим задачам из класса *NP*-трудных

В работе [28] представлен алгоритм построения автоматов-распознавателей по тестовым наборам, принципиально отличающийся от предлагаемых ранее алгоритмов для решения поставленной задачи. Отличие заключается в том, что данный алгоритм основан на сведении поставленной задачи к задаче о выполнимости булевой формулы (*Boolean satisfiability – SAT*). Алгоритм состоит из следующих пяти этапов:

1. построение префиксного дерева;
2. построение графа совместимости вершин префиксного дерева;
3. построение булевой КНФ-формулы, в случае наличия решения которой можно построить искомый автомат;
4. запуск стороннего программного средства для решения построенной КНФ-формулы (*SAT-solver*);
5. построение искомого автомата по решению, в случае его существования.

Первым шагом решения задачи, как и в алгоритмах слияния состояний *EDSM*, является построение префиксного дерева (*APTA*). В дальнейшем будем обозначать вершины дерева как V, допускающие вершины как V_+, недопускающие – как V_-. Для решения поставленной задачи необходимо найти в соответствие каждой вершине префиксного дерева одно из C состояний искомого автомата.

Вторым шагом алгоритма является построение графа совместимости префиксного дерева. Множество вершин данного графа совпадает с множеством вершин префиксного дерева, а две вершины соединены ребром, если они не могут соответствовать одному состоянию искомого автомата-распознавателя: при их слиянии нарушается свойство детерминированности. На рис. 5 приведен пример графа совместимости, построенного по префиксному дереву, приведенному на рис. 3.

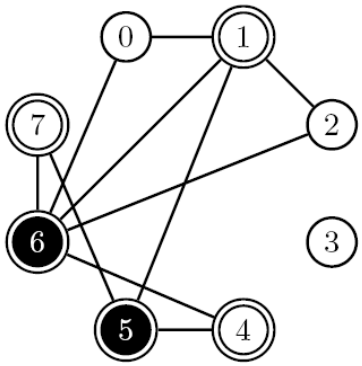

Рис. 5. Пример графа совместимости [28]

К примеру, ребро на рис. 5 соединяет вершины 0 и 1 потому, что их слияние приведет к слиянию (по путям *bb* и *abb*) вершин 5 и 7, что недопустимо. Граф совместимости строится методом динамического программирования: «ленивым» образом вычисляются несовместимые с *v* вершины исходя из вычисленных вершин, несовместимых с детьми *v*. Так как для решения задачи нам необходимо (но не достаточно) найти раскраску графа совместимости, будем говорить, что каждая вершина дерева сценариев раскрашена в один из *C* цветов.

Третьим, наиболее интересным шагом является построение булевой КНФ-формулы, по удовлетворяющему набору которой можно построить решение задачи (если такой существует). Авторы [28] предложили два способа построения данной формулы, ограничимся подробным описанием эффективного способа (*compact encoding*).

Опишем булевы переменные, которые будут использоваться для построения дизъюнктов формулы.

1. Основными переменными являются *переменные цвета* (*color variables*) $x_{v,i}$ для каждой вершины префиксного дерева *v* и каждого «цвета» *i* из *C*. Истинность переменной $x_{v,i}$ соответствует тому, что вершина *v* покрашена в цвет *i* (ей соответствует состояние *i* искомого автомата). Можно построить КНФ-формулу

лишь с использованием данных переменных (*direct encoding*), ее размер составит $O(|C|^2|V|^2)$.

2. *Переменные переходов* $y_{a,i,j}$ (*parent relation variables*) заданы для каждого символа алфавита a, каждой пары цветов i и j. Истинность переменной $y_{a,i,j}$ соответствует тому, что из каждой вершины цвета i переход искомого автомата по символу a ведет в вершину с цветом j. С использованием данных переменных размер КНФ-формулы можно сократить до $O(|C|^2|V|)$.
3. Для большего сокращения размера формулы вводятся *переменные допуска* (*accepting color variables*) z_i для каждого цвета i. Истинность данной переменной соответствует тому, что вершины префиксного дерева цвета i (состояние i искомого автомата) являются допускающими.

Заметим, что вспомогательные переменные y и z характеризуют лишь искомый автомат, не обращаясь при этом к префиксному дереву. С использованием описанных переменных построим КНФ-формулу, решение которой будет соответствовать корректной раскраске префиксного дерева. Данная формула состоит из следующих дизъюнктов.

1. $x_{v,1} \vee x_{v,2} \vee \ldots \vee x_{v,|C|}$ ($v \in V$) – каждой вершине v соответствует хотя бы один цвет.
2. $(\neg x_{v,i} \vee z_i) \wedge (\neg x_{w,i} \vee \neg z_i)$ ($v \in V_+$; $w \in V_-$; $i \in C$) – допускающие и недопускающие вершины не могут быть одного цвета i.
3. $y_{l(v),i,j} \vee \neg x_{p(v),i} \vee \neg x_{v,j}$ ($v \in V$; $i, j \in C$) – переменная переходов по символу $l(v)$ определена ($y_{l(v),i,j} = 1$), если определены цвета вершины v и его родителя $p(v)$.
4. $\neg y_{a,i,j} \vee \neg y_{a,i,h}$ ($a \in L$; $i, j, h \in C$; $j < h$) – существует не более одного перехода из состояния i по символу a.

Также в формулу добавляются дополнительные дизъюнкты, наличие которых упрощает работу программного средства, решающего задачу о выполнимости.

5. $\neg x_{v,i} \vee \neg x_{v,j}$ ($v \in V$, $i < j \in C$) – каждой вершине префиксного дерева соответствует не более одного цвета.

6. $y_{a,\,i,\,1} \vee y_{a,\,i,\,2} \vee \ldots \vee y_{a,\,i,\,|C|}$ $(a \in L;\, i \in C)$ – существует хотя бы один переход из состояния i по символу a.
7. $\neg y_{l(v),\,i,\,j} \vee \neg x_{p(v),\,i} \vee x_{v,\,j}$ $(v \in V;\, i, j \in C)$ – цвет вершины v определен, если определен цвет родителя $p(v)$ и переменная соответствующего перехода $y_{l(v),\,i,\,j}$.
8. $\neg x_{v,\,i} \vee \neg x_{w,\,i}$ $((v, w) \in E,\, i \in C)$ – цвета вершин, соединенных ребром графа совместимости, различны.

Четвертым шагом алгоритма является запуск программного средства для решения задачи о выполнимости на построенной КНФ-формуле. Отметим, что современные средства оперируют только с булевыми формулами в такой форме. Предложенное сведение строит формулу с больш*и*м числом дизъюнктов на некоторых задачах. Так, на ряде задач, предложенных в соревновании [31], размер формулы составляет более 100.000.000 дизъюнктов, в то время как современные программные средства справляются с порядка 5.000.000 дизъюнктов.

Для упрощения формулы авторы предложили перед построением КНФ-формулы выполнить на префиксном дереве несколько шагов алгоритма слияния состояний *EDSM*. Каждое слияние значительно сокращает размер префиксного дерева, что приводит к уменьшению формулы. Так как слияния производятся жадно, может быть произведено некорректное слияние. Однако, как утверждают авторы, на рассмотренных задачах несколько первых слияний не являются ошибочными.

Пятым, заключительным, шагом алгоритма является построение автомата по найденной выполняющей подстановке. Если такой подстановки не было найдено, то автомат по заданному обучающему словарю с заданным числом состояний C построить нельзя. Если же подстановка была найдена, то искомый автомат-распознаватель строится исходя из найденных значений переменных $x_{v,\,i}$.

Опишем теперь используемую авторами технику добавления *предикатов нарушения симметрии* (*symmetry breaking predicates*, *SBP*), идея которой заключается в следующем. Пусть мы пытаемся раскрасить некоторый граф в k цветов с помощью

средства решения задачи о выполнимости булевой формулы при том, что этого сделать нельзя. В таком случае для того, чтобы убедиться в отсутствии решения формулы, средство переберет $k!$ вариантов решения – по варианту на каждую перестановку цветов. Для предотвращения подобного на этапе преподсчета (в нашем случае после построения графа совместимости) фиксируются цвета вершин некоторой клики большого размера – это и есть предикаты нарушения симметрии. Так как задача нахождения наибольшей клики *NP*-полна, авторы закрепляют цвета большой клики, найденной с помощью приведенного в [28] жадного алгоритма.

Также авторы предложили следующий алгоритм решения задачи нахождения минимального автомата-распознавателя по заданному словарю:

1. найти клику K большого размера в графе совместимости;
2. инициализировать число цветов C размером найденной клики;
3. построить КНФ-формулу в соответствии с числом цветов C и предикатами нарушения симметрии для K;
4. решить данную КНФ-формулу;
5. если формула невыполнима, то добавить один цвет и вернуться к шагу 3;
6. вернуть автомат-распознаватель, соответствующий решению, найденному на шаге 4.

Экспериментальное исследование заключалось в том, что на задачах сравнивались предложенные авторами методы нахождения минимального автомата (со сведениями *direct encoding*, *compact encoding* и без дополнительных дизъюнктов) с лучшими алгоритмами слияния состояний *exbar* и *ed-beam* [30]. Результаты экспериментов приведены на рис. 6, при этом ось абсцисс соответствует экземплярам задачи, отсортированным по времени их решения.

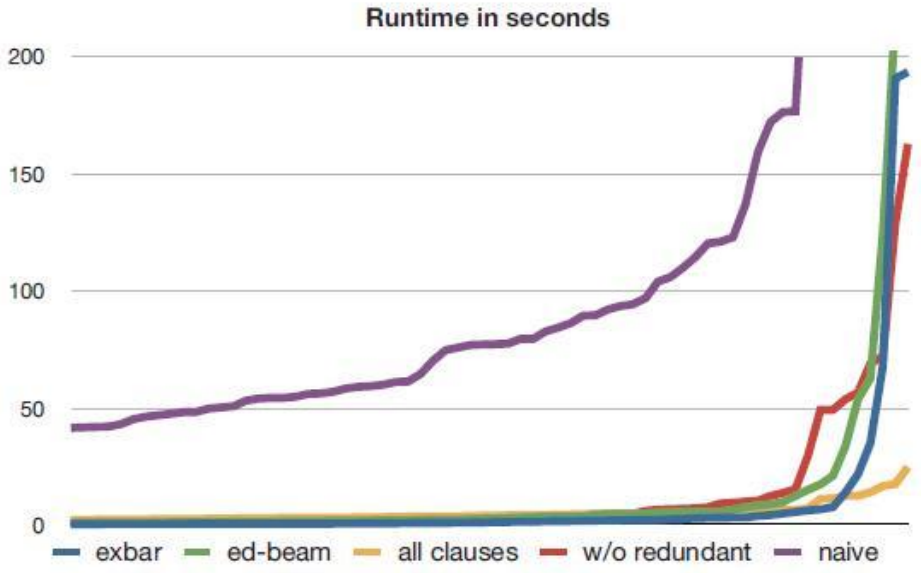

Рис. 6. Результаты вычислительных экспериментов, приведенные в работе [28]

Рассматривались задачи построения автоматов с числом состояний от 16 до 21; каждому запуску отводилось 200 секунд для решения; авторами использовалось средство *picosat*. Результаты показывают, что все методы, помимо наивного сведения, работают мгновенно на большинстве предложенных задач, однако на сложных задачах предложенный авторами метод (с дополнительными дизъюнктами) производительней остальных.

В работе [25] предлагается алгоритм сведения задачи построения автомата-распознавателя по словарям S^+ и S^- к задаче раскраски неориентированного графа, другой классической *NP*-полной задаче.

1.1.3. Построение конечных преобразователей по обучающим примерам

Данный пункт соответствует указанному в [14]. Одной из работ по конечным автоматам-преобразователям является работа [36]. Так же, как и в работах [34] и [35], посвященных построению конечных автоматов-распознавателей, в работе [36]

применяется (1+1) эволюционная стратегия. Конечный преобразователь представляется в виде двух таблиц – таблицы значений функции переходов и таблицы значений функции выходов. При этом предполагалось, что на каждом из переходов конечный преобразователь может вывести не более одного символа. Входными данными для построения конечного автомата-преобразователя является набор обучающих примеров – пар слов, одно из которых является входным, а второе – соответствующим ему выходным.

Опишем алгоритм выполнения операции мутации в алгоритме, предложенном в рассматриваемой работе. Выполняется мутация одного из трех типов:

- добавление состояния – выполняется с вероятностью 0,1, если автомат содержит меньше заданного максимального числа состояний. При этом переходы из нового состояния генерируются случайным образом;
- удаление состояния;
- случайное изменение одного перехода.

Последние две мутации из перечисленных выбираются равновероятно.

В работе [36] рассматриваются три варианта функции приспособленности:

- на основе строгого сравнения (*strict*);
- на основе вычисления расстояния Хэмминга [27] (*hamming*);
- на основе вычисления редакционного расстояния (расстояния Левенштейна) [6] (*edit*).

1.1.4. Построение управляющих автоматов по обучающим примерам

Методы построения управляющих автоматов по обучающим примерам разрабатываются, в основном, на кафедре «Компьютерные технологии» НИУ ИТМО. Вкратце приведем основные результаты исследований.

Цель работы [3] заключалась в разработке метода совместного применения генетического программирования и верификации моделей для построения автоматов управления системами со сложным поведением. Исходными данными являлись тесты (каждый тест состоит из входной последовательности событий и соответствующей ей последовательности выходных действий, которую должен вырабатывать автомат) для системы со сложным поведением и утверждений на языке логики линейного времени (*Linear Time Logic, LTL*). В предыдущей работе [3] построение автоматов осуществлялось только на основе тестов. Одним из недостатков такого подхода является то, что с помощью тестов достаточно трудно (такое описание будет слишком громоздким) описать все варианты поведения. Это означает, что построенный таким образом автомат нельзя использовать без дополнительных проверок. В случае обнаружения ошибок в построенном автомате его придется модифицировать вручную, что достаточно трудно, так как зачастую компьютер выделяет состояния и переходы не так, как это делает человек.

Предлагаемый в [15] подход исправляет указанный недостаток. Функция приспособленности, используемая в алгоритме генетического программирования, учитывает успешность прохождения тестов и истинность *LTL*-формул. В случае прохождения всех тестов и выполнения всех *LTL*-формул можно считать, что автомат с заранее заданным поведением построен. На одной из задач применение верификации позволяет построить удовлетворяющий спецификации автомат управления дверями лифта, который не получается построить только на основе тестов, а на другой применение верификации замедляет построение автомата.

В работе [19] был разработан и реализован метод генерации конечных автоматов по заданной функции приспособленности, основанный на *муравьином алгоритме*. Эффективность разработанного метода была оценена путем сравнения с генетическими алгоритмами для задачи об «Умном муравье» и задаче о генерации автоматов на основе тестовых примеров. Эксперименты показали, что среднее время работы

предлагаемого в [19] алгоритма для построения целевых автоматов в этих задачах в несколько раз меньше времени работы генетического алгоритма.

В работе [1] для генерации автоматов управления объектами со сложным поведением предлагается применять генетическое программирование. При этом вместо подхода [8], в котором для оценки качества управляющего автомата используется моделирование, занимающее обычно большое время, применяется подход, в котором выполняется сравнение поведения автоматов с поведением, обеспечиваемым за счет управления человеком. Особенность рассматриваемого подхода состоит в том, что он позволяет использовать объекты управления не только с дискретными, но и с *непрерывными параметрами*. Применение подхода иллюстрируется на примере создания автомата, управляющего моделью самолета в режиме «мертвая петля» (рис. 7).

Рис. 7. Выполнение моделью самолета фигуры «мертвая петля»

В статье [18] описывается модификация метода построения управляющих конечных автоматов по обучающим примерам, или тестам. В предложенной

модификации построение автоматов осуществляется на основе муравьиных алгоритмов и эволюционных стратегий. Данные алгоритмы поисковой оптимизации улучшили производительность метода построения автоматов, что было показано в их сравнении с генетическим алгоритмом [1]. В большинстве случаев построенные автоматы справляются со своей задачей, однако допускают небольшие ошибки при выполнении фигур пилотажа.

В работе [11] описывается алгоритм, основанный на методах решения задачи о выполнимости булевой формулы и предназначенный для построения управляющих конечных автоматов, в отличие от работы [28], где предложен алгоритм построения конечных автоматов-преобразователей.

Входными данными для описанного в работе метода машинного обучения является набор сценариев, задающих поведение управляющего автомата. Каждый сценарий является последовательностью троек $T_1, \ldots, T_n$, $T_i = \langle e_i, f_i, A_i \rangle$, где e_i – входное событие, f_i – булева формула от входных переменных, задающая охранное условие, A_i – последовательность выходных воздействий.

Первым этапом метода является построение дерева сценариев из имеющегося набора сценариев. Построение дерева сценариев осуществляется за время $O(n^2)$. Остальные этапы метода аналогичны соответствующим этапам метода, предложенного в [28]. В рамках второго этапа на основе построенного дерева сценариев осуществляется построение графа совместимости, после чего в рамках третьего этапа метода строится булева КНФ-формула, задающая ограничения на раскраску графа совместимости.

Последним шагом рассматриваемого метода машинного обучения является запуск стороннего средства, находящего выполняющую подстановку для заданной булевой КНФ-формулы и построение искомого управляющего автомата. В качестве используемого программного средства выбрана программа *cryptominisat*, признанная победителем соревнования *Sat Race 2010* [41].

1.2. Программные средства для решения задачи удовлетворения ограничений

Программные средства для решения задачи удовлетворения ограничений можно разделить на две категории.

1. Программные средства, решающие задачу напрямую, используя такие техники, как поиск с возвратом и распространение ограничений.
2. Программные средства, сводящие задачу удовлетворения ограничений к задаче выполнимости булевой формулы (*SAT*).

Ниже мы рассмотрим программные средства, использующие и тот, и другой подход к решению задачи. Отметим, что в общем случае задача удовлетворения ограничений не разрешима за полиномиальное время.

1.2.1. Программное средство *Choco*

Программное средство *Choco* [21] относится к средствам, решающим задачу удовлетворения ограничений напрямую, и представляет собой библиотеку для языка программирования *Java*. Экземпляр задачи удовлетворения ограничений задается на самом языке *Java* путем указания переменных (целочисленных, вещественных или перечислений), их областей значений и ограничений на них. В дальнейшем предполагается использование именно этого программного средства.

Приведем небольшой фрагмент кода, задающий часть ограничений для «магического квадрата», классической задачи удовлетворения ограничений:

```
CPModel m = new CPModel();
IntegerVariable[][] var = new IntegerVariable[n][n];
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
    var[i][j] = Choco.makeIntVar("var_" + i + "_" + j, 1, n * n);
    m.addVariable(var[i][j]);
}
```

```
for (int i = 0; i < n * n; i++) {
    for (int j = i + 1; j < n * n; j++) {
        Constraint c = (Choco.neq(var[i / n][i % n], var[j / n][j % n]));
        m.addConstraint(c);
    }
}
```

В данном примере создается по переменной для каждой клетки «магического квадрата» (рассматривается квадрат размера $n \times n$). Далее, для каждой пары клеток квадрата задается ограничение на неравенство чисел в этих клетках.

Choco поддерживает следующие ограничения: равенство переменных, неравенство переменных, операции отношения, равенство или неравенство знака переменных, следование или эквивалентность условий на переменные и некоторые другие. Для задания ограничений могут использоваться такие операции, как сумма, произведение, взятие модуля и т.д.

Рассмотрим принципы, на основе которых *Choco* решает задачу удовлетворения ограничений. Принципы, которые будут рассмотрены, используются и в других программных средства, решающих задачу удовлетворения ограничений напрямую.

Поиск с возвратом (backtracking). Принцип заключается в рекурсивном переборе возможных значений переменной. Выбирается одна из переменных *x*, значение которой в данный момент не определено, после чего ей присваивается одно их возможных значений, на основе чего упрощаются ограничения. Если программным средством будет установлено, что при данном значении переменной *x* задача неразрешима, оно вернется к моменту выбора значения *x* и попробует другое.

Choco предоставляет пользователю возможность как использования стандартных стратегий перебора переменных и значений, так и определения своих собственные.

Распространение ограничений (constraint propagation). Распространение ограничений – способ отсечь заведомо недопустимые значения переменных на основе

анализа ограничений. Использование такого подхода более эффективно, чем перебор всех возможных значений переменных.

В *Choco* распространение ограничений реализовано на основе событийного подхода: поддерживается очередь событий двух типов. Каждое событие первого типа связано с некоторой переменной и при «исполнении» сужает область ее значений. При «исполнении» какого-либо события на переменной *x* ограничения, связанные с переменной *x*, вновь анализируются, что приводит к появлению новых событий, которые тут же добавляются в очередь. События второго типа связаны с ограничениями и являются вызовами методов их обработки. Схема реализации метода распространения ограничений в *Choco* приведена на рис. 8 [20].

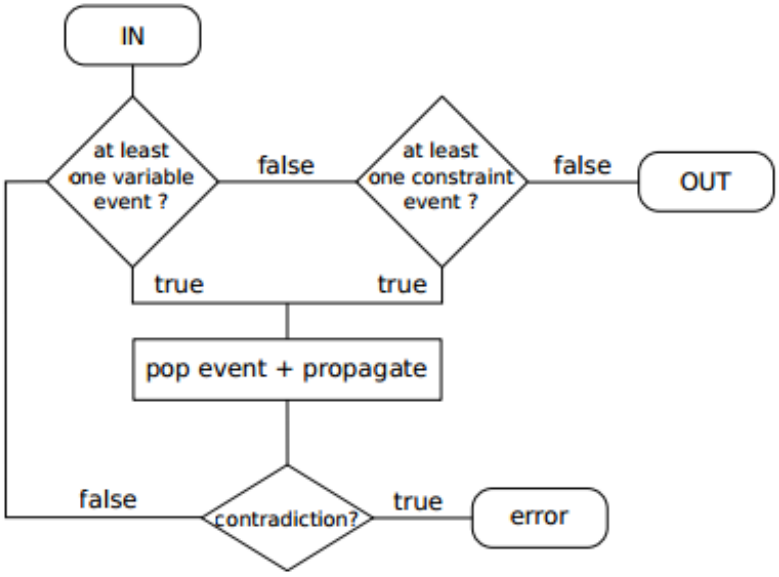

Рис. 8. Распространение ограничений в *Choco* [20]

Программное средство *Choco* участвовало в соревнованиях *International CSP Solver Competition* и в 2009 году заняло первые места в категориях *2-ARY-INT* и *Alldiff+Cumul+Elt+WSum* этого соревнования [24].

1.2.2. Программное средство *Mistral*

Программное средство *Mistral* [38] представляет собой библиотеку для языка программирования *C++*. Способ работы с ним аналогичен способу работы с программным средством *Choco*. Перечислим отличительные особенности программного средства *Mistral*.

- Для целочисленных переменных используются различные формы внутреннего представления допустимых значений, в том числе битовые векторы и списки значений.
- Используется техника *вложенных предикатов*, в рамках которой создаются ограничения для каждого подвыражения предиката. Поясним это на примере. Рассмотрим ограничение $x_0 \cdot x_1 + x_2 = x_3$, где $x_0, \ldots, x_3$ – переменные. Оно будет разделено на ограничения $x_0 \cdot x_1 = y_0$, $y_0 + x_2 = y_1$, $y_1 = x_3$, где y_0 и y_1 – дополнительные переменные.
- Поддерживается ограничение *NValue*, которое определяет число различных значений в списке целых чисел и является обобщением ограничения *AllDifferent*.

Программное средство *Mistral* заняло первые места в нескольких категориях соревнования *International CSP Solver Competition* 2009 [24].

1.2.3. Программное средство *Sugar*

Sugar [43] относится к программным средствам, решающим задачу удовлетворения ограничений посредством ее сведения к задаче выполнимости булевой формулы (*SAT*) и запуска программного средства для решения этой задачи (*zChaff*, *minisat*, *CryptoMiniSat* и т.д.).

Входные данные для *Sugar* задаются во входном файле, при этом ограничения записываются в префиксной нотации. Файл должен состоять из описаний переменных

и ограничений на них. Приведем пример входного файла для программного средства *Sugar*:

```
(int q_1 1 8)
(int q_2 1 8)
(int q_3 1 8)
(int q_4 1 8)
(int q_5 1 8)
(int q_6 1 8)
(int q_7 1 8)
(int q_8 1 8)
(alldifferent q_1 q_2 q_3 q_4 q_5 q_6 q_7 q_8)
(alldifferent (+ q_1 1) (+ q_2 2) (+ q_3 3) (+ q_4 4)
     (+ q_5 5) (+ q_6 6) (+ q_7 7) (+ q_8 8))
(alldifferent (- q_1 1) (- q_2 2) (- q_3 3) (- q_4 4)
    (- q_5 5) (- q_6 6) (- q_7 7) (- q_8 8))
```

При сведении задачи удовлетворения ограничений к задаче выполнимости булевой формулы используется следующий подход. Для каждой целочисленной переменной x и для каждого ее возможного значения a вводятся булевы переменные, задающие ограничения вида $x \leq a$ (или $x \geq a$). Такой способ сведения задачи удовлетворения ограничений к задаче выполнимости булевой формулы называется *кодированием порядка* (order encoding).

Отметим, что альтернативными способами кодирования целочисленных переменных являются кодирование каждого бита числа как булевой переменной (бинарное кодирование) и кодирование в виде булевых переменных утверждений вида $x = a$ (прямое кодирование). Пусть имеется переменная x с областью значений $\{1, 2, 3, 4, 5\}$. В этом случае переменной x будут соответствовать следующие наборы значений булевых переменных:

- бинарное кодирование: [1, 0, 1];

- прямое кодирование: [0, 0, 1, 0, 0];
- кодирование порядка: [0, 0, 1, 1, 1].

Программное средство *Sugar* стало победителем в категории GLOBAL соревнований *International CSP Solver Competition* 2008 и 2009 года [23, 24]. При этом средство *Sugar* использовалось совместно с программами *minisat* и *picosat*, находящими выполняющую подстановку для заданной булевой формулы.

1.2.4. Программное средство *Bee*

Bee (Ben-Gurion-University Equi-propagation Encoder) [37] – программное средство для преобразования экземпляров задачи удовлетворения ограничений с конечными областями значений переменных в КНФ-формулы, написанное на языке *Prolog*. Полученные КНФ-формулы могут быть решены одним из средств, предназначенных для решения задачи выполнимости булевой формулы. Для представления целочисленных переменных в виде булевых переменных используется кодирование порядка, как и в средстве *Sugar*.

В основе работы *Bee* лежит идея вывода следствий из заданных ограничений (*equi-propagation*), за счет которых ограничения могут быть упрощены. Приведем пример. Пусть в результате анализа ограничений было установлено, что должно выполняться утверждение $x = -y$, где x и y – переменные. В этом случае одну из этих переменных можно исключить из всех ограничений, заменив ее другой. Сама процедура вывода следствий реализована в программном средстве на основе двоичных диаграмм решений.

1.2.5. Программное средство *Sat4j*

Sat4j [42] – программное средство, способное решать как задачу удовлетворения ограничений, так и задачу выполнимости булевой формулы, при этом решение первой задачи сводится к решению второй. Средство *Sat4j* написано на языке *Java* и запускается командой вида:

```
java -jar sat4j-csp.jar cspfile.xml
```

Здесь `cspfile.xml` – файл, в котором экземпляр задачи удовлетворения ограничений записан в формате *XML CSP format 2.0*, разработанном для второго соревнования *International CSP Solver Competition*, проведенного в 2006 году.

Разработчики подчеркивают гибкость и удобство данного программного средства, но не гарантируют быстроту его работы.

1.2.6. Сравнение рассмотренных программных средств

В табл. 1 приведено сравнение рассмотренных в настоящем обзоре программные средств для решения задачи удовлетворения ограничений. Сравнение осуществляется по следующим критериям:

- метод решения задачи;
- формат входных данных;
- язык программирования, на котором написано программное средство;
- особенности программного средства.

Таблица 1. Сравнение рассмотренных в обзоре программных средств для решения задачи удовлетворения ограничений

Название программного средства	Метод решения задачи *CSP*	Способ задания ограничений	Язык программирования	Особенности программного средства
Choco	Прямое решение	Код на языке *Java*	*Java*	Поддержка вещественных переменных, поддержка большого числа ограничений, возможность создавать пользовательские стратегии перебора переменных и их значений
Mistral	Прямое решение	Код на языке *C++*	*C++*	Различные способы представления областей значений переменных, разнообразие используемых эвристик
Sugar	Сведение к задаче выполнимости булевой формулы	Файл в специфическом для средства формате или в формате *XCSP 2.0*	*Java*	Кодирование порядка, возможность решать задачи оптимизации с ограничениями
Bee	Сведение к задаче выполнимости булевой формулы	Файл в специфическом для средства формате	*Prolog*	Кодирование порядка, метод *equi-propagation*
Sat4j	Сведение к задаче выполнимости булевой формулы	Файл в формате *XCSP 2.0*	*Java*	Включает в себя средство для решения задачи выполнимости булевой формулы

2. ПОСТАНОВКА ЗАДАЧИ ПОСТРОЕНИЯ УПРАВЛЯЮЩИХ АВТОМАТОВ ПО СЦЕНАРИЯМ РАБОТЫ ПРОГРАММЫ И ИССЛЕДОВАНИЕ ЕЕ СЛОЖНОСТИ

В настоящем разделе приводится формальная постановка задачи построения управляющих автоматов по сценариям работы программы. Приводится доказательство принадлежности поставленной задачи классу *NP*-трудных, условие принадлежности классу *NP* рассматриваемой задачи.

2.1. Постановка задачи построения управляющего автомата по сценариям работы

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 тот соответствующий ему переход, для которого охранное условие оказывается истинным. 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 недопускающие. Пример управляющего автомата приведен на рис. 9.

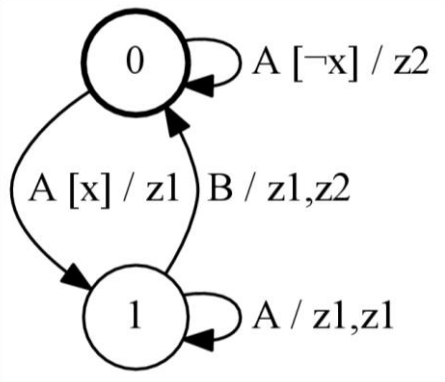

Рис. 9. Пример управляющего конечного автомата

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.

Так, например, автомат, приведенный на рис. 9, удовлетворяет следующим сценариям работы:

- $\langle A; \neg x; z2 \rangle, \langle A; x; z1 \rangle, \langle A; true; z1,z1 \rangle$;
- $\langle A; x; z1 \rangle, \langle B; true; z1,z2 \rangle, \langle A; x; z1 \rangle$;

- <*A*; *x*; *z*1>, <*B*; *true*; *z*1,*z*2>, <*A*; ¬*x*; *z*2>, <*A*; ¬*x*; *z*2>.

К искомой автоматной модели обязательно предъявляется требование непротиворечивости – не должно быть двух переходов, исходящих из одного состояния управляющего автомата и одновременно выполнимых при некоторой комбинации события и входных переменных. В некоторых системах к автоматам управления предъявляется требование *полноты* – любой комбинации события и входных переменных должен соответствовать переход в каждом состоянии.

При этом можно выделить два вида требования полноты. «Сильная» полнота подразумевает то, что для каждого состояния найдется инцидентный переход с любым событием, встречающимся во входных данных (сценариях работы). Пример такого автомата приведен на рис. 2. «Слабая» полнота накладывает ограничение на охранные условия: если состоянию инцидентен хотя бы один переход с данным событием *E*, то при любой комбинации значений входных переменных данному состоянию должен быть инцидентен переход, помеченный *E* и охранным условием, выполнимым при данной комбинации переменных. Пример автомата, удовлетворяющего требованию «слабой» полноты, приведен на рис. 9. В дальнейшем под требованием полноты будем иметь ввиду «слабую» полноту.

Решается задача построения управляющего конечного автомата, удовлетворяющего следующим требованиям:

- число состояний автомата равно заданному числу *C*;
- автомат удовлетворяет заданному множеству сценариев работы *Sc*;
- автомат удовлетворяет требованию полноты;
- каждый переход автомата подтвержден хотя бы одним сценарием работы.

Без последнего требования можно эвристически дополнить автомат до полного, однако поведение автомата на добавленных переходах ничем не подтверждено.

2.2. Доказательство принадлежности поставленной задачи классу *NP*-трудных

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.

2.2.1. Используемые понятия теории сложности

Приведем понятия, используемые при проведении теоретической оценки сложности поставленной задачи. Данные понятия подробно изложены в книге [13].

Классом $DTIME(f(n))$ называется множество языков, для которых существует детерминированная 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. То есть,

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

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

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

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 [13], классы Σ_1 и NP равны, следовательно, определение Σ_1 является альтернативным определением класса NP.

Приведем формальное определение остальных классов полиномиальной иерархии Σ_i:

$$\Sigma_i = \{L \mid \exists R(x, y_1, \ldots, y_i) \in P, p : \forall x \in L \Leftrightarrow \exists y_1 \forall y_2 \exists y_3 \ldots Q y_i : \forall j \mid y_j \mid \leq p(\mid x \mid), R(x, y_1, \ldots, y_i)\},$$

где L – формальный язык, p – полином, Q соответствует квантору существования, если i нечетно, и квантору всеобщности, если i четно.

Введем понятие *сведения 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 он является NP-трудным и принадлежит классу NP.

2.2.2. Доказательство принадлежности задачи удовлетворения ограничений 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 [26] 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^+ в 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, для каждого допускающего состояния добавим помеченный тройкой <$, *true*, (*accept*)> переход, начинающийся и заканчивающийся в этом состоянии. 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.

2.3. Условие принадлежности рассматриваемой задачи классу *NP*

Докажем принадлежность задачи построения управляющего автомата по сценариям работы классу *NP*. 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 как $|Sc|$ обозначена 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, было показано существование полиномиального отношения *R*, что доказывает принадлежность исследуемой задачи классу *NP*. Однако для того, чтобы выполнить для элемента сценария проверку, удовлетворяет ли данному элементу автомат, необходимо проверить на равенство булевы формулы, которыми задаются охранные условия. В общем случае, если охранные условия заданы строковыми представлениями, такую проверку невозможно выполнить за полиномиальное время [13]. Если же охранные условия заданы таблицами истинности, или же охранные условия зависят от ограниченного числа переменных, то проверку на равенство можно выполнить за полиномиальное время, что обеспечивает принадлежность поставленной задачи классу *NP*.

3. РАЗРАБОТКА И РЕАЛИЗАЦИЯ МЕТОДА ПОСТРОЕНИЯ УПРАВЛЯЮЩИХ АВТОМАТОВ

В настоящем разделе приводится описание алгоритма и программной реализации метода построения управляющих автоматов по сценариям работы программы.

3.1. Алгоритм построения управляющих автоматов

В настоящем подразделе приводится алгоритм построения управляющих автоматов по безошибочным сценариям работы программы. Предлагаемый алгоритм включает в себя пять основных этапов.

1. Построение дерева сценариев.
2. Построение графа совместимости вершин дерева сценариев.
3. Построение набора ограничений на целочисленные переменные, задающего требования к «раскраске» построенного графа и выражающей непротиворечивость и полноту системы переходов искомого автомата.
4. Запуск сторонней программы, решающей задачу удовлетворения построенный ограничениям (*constraint satisfaction problem – CSP*), или вызов соответствующего метода сторонней библиотеки.
5. Построение автомата по найденной выполняющей подстановке.

3.1.1. 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, задающей ту же булеву функцию, что и f_i. 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.

Рассмотрим пример работы алгоритма построения дерева сценариев. Пусть набор сценариев состоит из следующих последовательностей:

- $<T; x; z1>, <T; \neg x; z2>, <T; \neg x; z3>, <T; x; z1>$;
- $<T; \neg x; z2>, <T; \neg x; z3>, <T; x; z1>, <T; x; z1>$;

- <*T*; *x*; *z*1>, <*T*; *x*; *z*1>, <*T*; ¬*x*; *z*2>;
- <*T*; ¬*x*; *z*2>, <*T*; ¬*x*; *z*3>, <*T*; *x*; *z*1>, <*T*; ¬*x*; *z*2>.

Данные сценарии работы содержат единственное входное событие *T* (например, оно соответствует тику часов), единственную переменную *x*, два различных охранных условия *x* и ¬*x*, три различных последовательности выходных воздействий (*z*1), (*z*2), (*z*3). Построенное описанным алгоритмом дерево сценариев приведено на рис. 10.

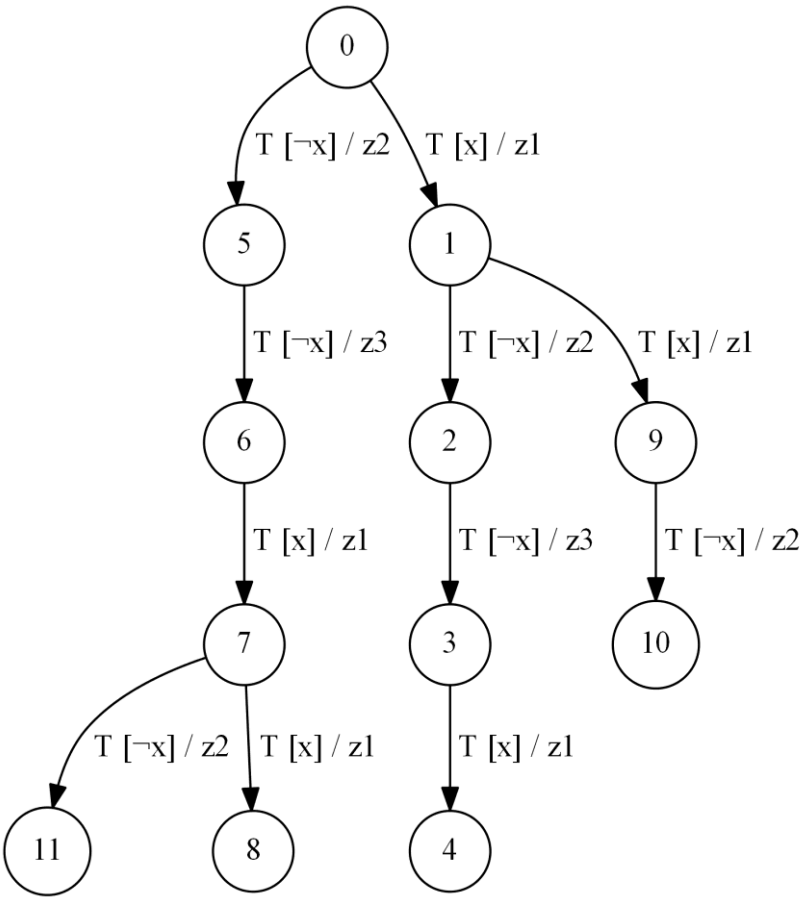

Рис. 10. Пример дерева сценариев

3.1.2. 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 [4, 10].

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; $values_1$>…<e_k; $values_k$>, различающая вершины x и y, а вершины v и u различает последовательность <e; *values*> <e_1; $values_1$>…<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.

Для дерева сценариев, приведенного на рис. 10, описанный алгоритм построит граф совместимости, приведенный на рис. 11. На рисунке сплошными линиями выделены ребра графа совместимости, а штриховыми – исходные ребра дерева сценариев. Вершины с номерами 2 и 5 соединены с вершинами 0, 1, 7 и 9 так как их различает последовательность <*T*; *x* = *false*> – пометки соответствующих ребер дерева сценариев различаются в выходных воздействиях. Других различающих последовательностей в приведенном дереве сценариев не существует.

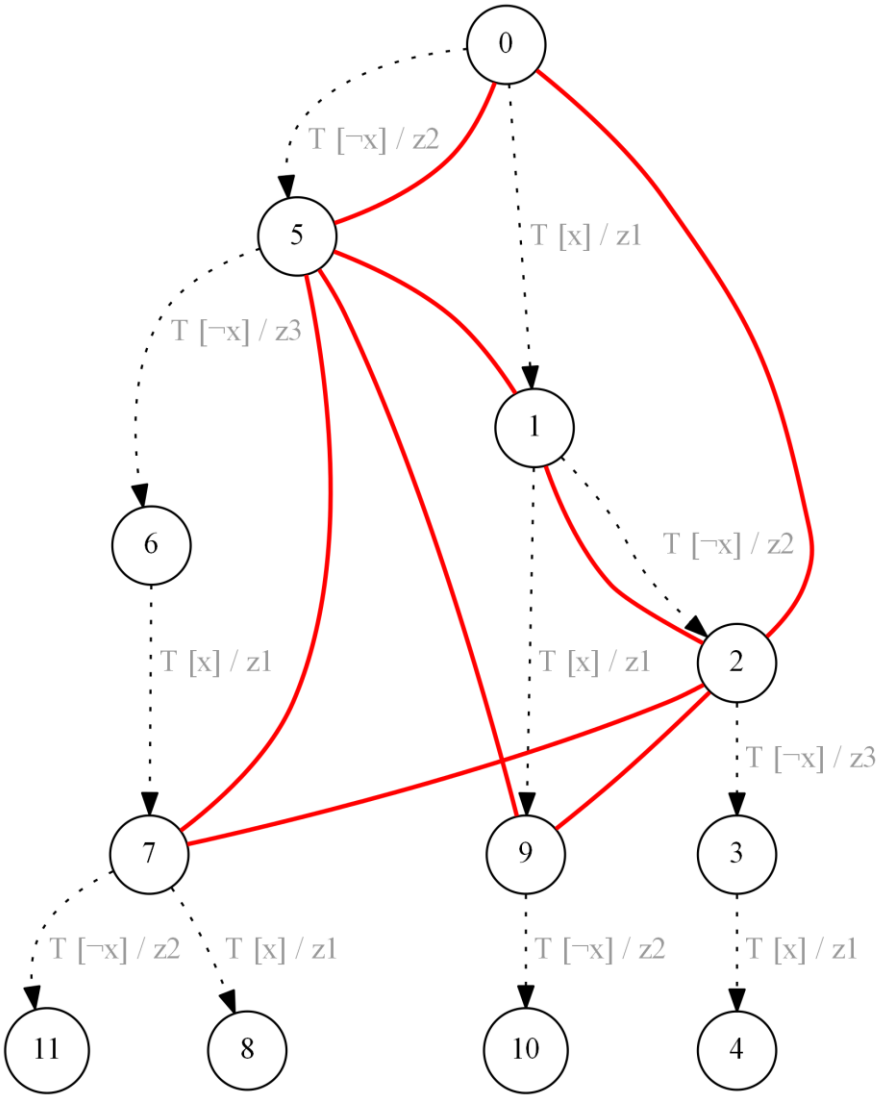

Рис. 11. Пример графа совместимости

3.1.3. Построение набора ограничений на целочисленные переменные

Следующим этапом работы алгоритма построения управляющих конечных автоматов по сценариям работы программы является построение набора ограничений на целочисленные переменные, задающего требования к «раскраске» графа совместимости и выражающего непротиворечивость и полноту системы переходов искомого автомата. Напомним, что на вход алгоритму подается число C состояний результирующего автомата.

Будем в дальнейшем множество всех условий переходов, встречающихся в сценариях работы Sc обозначать символом F. Множество входных событий e будем обозначать как E. Под F_e будем подразумевать множество условий переходов, помеченных входных событием e.

Для построения набора ограничений создадим, и будем использовать следующие целочисленные переменные.

1. Переменные x_v соответствуют цвету каждой вершины дерева сценариев v и принимают значения от 0 до $C-1$. Напомним, что вершины одного цвета будут объединены в одно состояние результирующего автомата.
2. Переменные $y_{i,e,f}$ являются вспомогательными для построения ограничений, задающих непротиворечивость (детерминированность) искомого управляющего автомата, и хранят в себе информацию о его переходах. Данные переменные являются вспомогательными, так как наличие в результирующем автомате переходов не определяется исходя из их значений. Используются переменные для каждого состояния i результирующего автомата (значение от 0 до $C-1$), каждого события e, каждого условия перехода f из множества F_e, встречающегося в сценариях. Каждая переменная принимает значения от 0 до $C-1$ и соответствует

номеру состояния, в которое ведет переход искомого автомата из состояния i по событию e и условию перехода f.

3. Переменные $z_{i,e,f}$ используются для задания требования полноты искомого автомата и принимают значения 0 или 1, то есть по своей сути являются логическими. Данные переменные задаются для каждого состояния i результирующего автомата (значение от 0 до $C - 1$), каждого события e, каждого условия перехода f из F_e, встречающегося в сценариях. Переменная $z_{i,e,f}$ равна 1, если существует вершина v в дереве сценариев, цвет которой равен i ($x_v = i$), и из нее ведет ребро, помеченное событием e и условием перехода f. В противном случае значение переменной равно 0. Таким образом, данные переменные хранят информацию о структуре переходов результирующего автомата, получающегося в результате объединения вершин дерева сценариев.

Составим набор ограничений на указанные переменные, задающий требования полноты и непротиворечивости искомого автомата.

1. $x_0 = 0$ – ограничение, задающее соответствие корня дерева сценариев начальному состоянию искомого автомата. В настоящем методе начальным состоянием автомата считается состояние с номером 0.
2. $x_v \neq x_u$ (для каждой несовместимой пары вершин дерева сценариев u и v, то есть соединенных ребром в графе совместимости) – ограничения, задающие непротиворечивость искомого автомата. Они гарантируют отсутствие различающих последовательностей, ведущих из одного состояния автомата. Число ограничений данного вида равно числу ребер графа совместимости, то есть в худшем случае таких ограничений может быть $O(n^2)$, где n – число вершин дерева сценариев.
3. $(x_v = i) \Rightarrow (x_u = y_{i,e,f})$ (для каждого цвета i и каждого ребра дерева сценариев vu, помеченного событием e и условием перехода f) –

ограничения, задающие детерминированность искомого автомата. А именно, если вершине v присвоен цвет i, то цвет вершины u совпадает со значением переменной $y_{i,e,f}$, хранящей номер состояния автомата, в которое ведет переход из состояния i, помеченный событием e и условием перехода f. Число данных ограничений равно $C\cdot(n-1)$.

4. $z_{i,e,f} = 1 \Leftrightarrow (x_{v_1} = i \vee \ldots \vee x_{v_n} = i)$ (для каждого цвета i, входного события e, условия перехода f из F_e, встречающегося в заданных сценариях, и вершин $v_1 \ldots v_n$ дерева сценариев, из которых ведет ребро, помеченное событием e и условием перехода f) – ограничения, необходимые для правильного задания значений переменных $z_{i,e,f}$. Количество данных ограничений оценивается как $O(C\cdot|E|\cdot|F|)$, где как $|E|$ обозначено число событий, а как $|F|$ – число различных условий перехода.

5. $\left(\sum_{f \in F_e}(z_{i,e,f}\cdot c(f)) = 0\right) \vee \left(\sum_{f \in F_e}(z_{i,e,f}\cdot c(f)) = 2^m\right)$ (для каждого цвета i и каждого события e) – ограничения, задающие требование полноты искомого автомата. Здесь как $c(f)$ обозначена функция, которая возвращает число выполняющих подстановок для булевой формулы f. При подсчете $c(f)$ считается, что булева формула зависит от всех m переменных, содержащихся в сценариях (например, $c(true) = 2^m$, а $c(x_1 \vee \neg x_2) = 2^{m-2}$). Сумма $\sum_{f \in F_e}(z_{i,e,f}\cdot c(f))$ равна числу комбинаций значений входных переменных *values*, для которых существует переход из состояния i, помеченный событием e и условием перехода f таким, что выполняется $f(values)$. Условие полноты (в «слабом» смысле) искомого автомата выражается тем, что или для любого значения входных переменных найдется переход, или ни для одного из значений переменных перехода не существует. Заметим, что условие того, что для любого

значения входных переменных найдется переход, можно выразить как $\sum_{f \in F_e} (z_{i,e,f} \cdot c(f)) = 2^m$, так как считается, что требование непротиворечивости уже выполнено, то есть все формулы, для которых выполняется $z_{i,e,f} = 1$, попарно не имеют общих выполняющих подстановок. Для удовлетворения требованию «сильной» полноты (см. выше) можно оставить в ограничении лишь второй конъюнкт.

Приведенные ограничения пяти типов составляют набор, задающий требования непротиворечивости и полноты искомого автомата, удовлетворяющего сценариям *Sc* и содержащего *C* управляющих состояний. Пример набора ограничений приведен в приложении А.

3.1.4. Нахождение выполняющей подстановки для построенного набора ограничений

Следующим этапом работы алгоритма является нахождение выполняющей подстановки для построенного на предыдущем этапе набора ограничений на целочисленные переменные. Для решения задачи удовлетворения ограничений используется библиотека *Choco* [21] для языка программирования *Java*. Данная библиотека предоставляет удобный интерфейс пользователя для создания переменных и ограничений, а также реализует производительные методы решения задачи удовлетворения ограничений. Программная реализация будет подробнее описана в следующем подразделе.

3.1.5. Построение управляющего автомата по найденной выполняющей подстановке

Если библиотека *Choco* не обнаружила набор значений переменных, удовлетворяющих построенным ограничениям, то будем считать, что по данному набору сценариев *Sc* невозможно построить управляющий автомат с заданным числом

состояний *C*. Если же программа обнаружила выполняющий набор, то построим искомый автомат. Для этого на основании полученных значений переменных x_v определим цвет каждой вершины дерева сценариев. На рис. 12 приведен пример раскраски дерева сценариев, представленного на рис. 10.

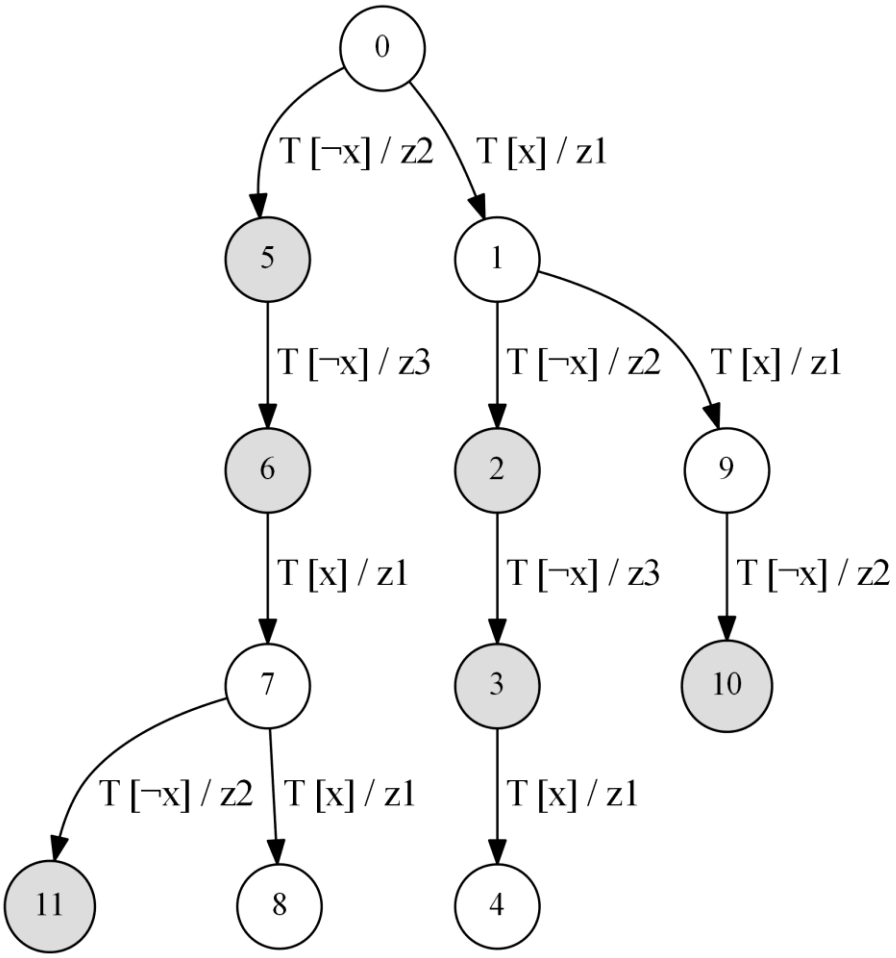

Рис. 12. Пример раскраски дерева сценариев

После этого объединим все вершины одного цвета в одно состояние автомата, а начальным состоянием положим состояние, соответствующее цвету корня дерева сценариев. Множество исходящих из состояния переходов построим из объединения множеств ребер, исходящих из вершин заданного цвета. Например, после объединения вершин дерева, приведенного на рис. 12, получим автомат, приведенный на рис. 13.

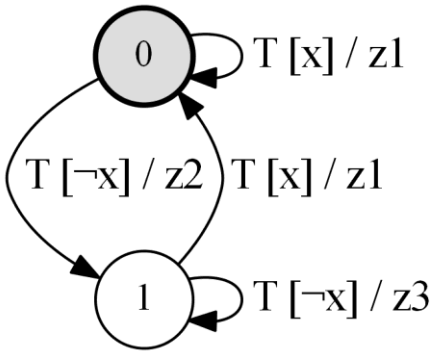

Рис. 13. Автомат, полученный после объединения вершин дерева сценариев

Отметим, что не исключено существование нескольких автоматов, удовлетворяющих сценариям множества *Sc*. Также отметим, что обозначенным сценариям работы удовлетворяет автомат, приведенный на рис. 14.

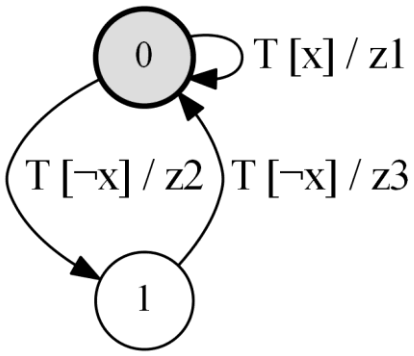

Рис. 14. Автомат, не обладающий свойством полноты

Данный управляющий автомат обладает свойством непротиворечивости, но не обладает свойством полноты. Такой автомат можно получить, если не добавлять ограничения четвертого и пятого типов.

3.2. Программная реализация метода построения управляющих автоматов

Программная реализация разработанного метода построения управляющих автоматов по безошибочным сценариям работы выполнялась на языке

программирования *Java*. Основные классы и связи между ними представлены на схеме (рис. 15).

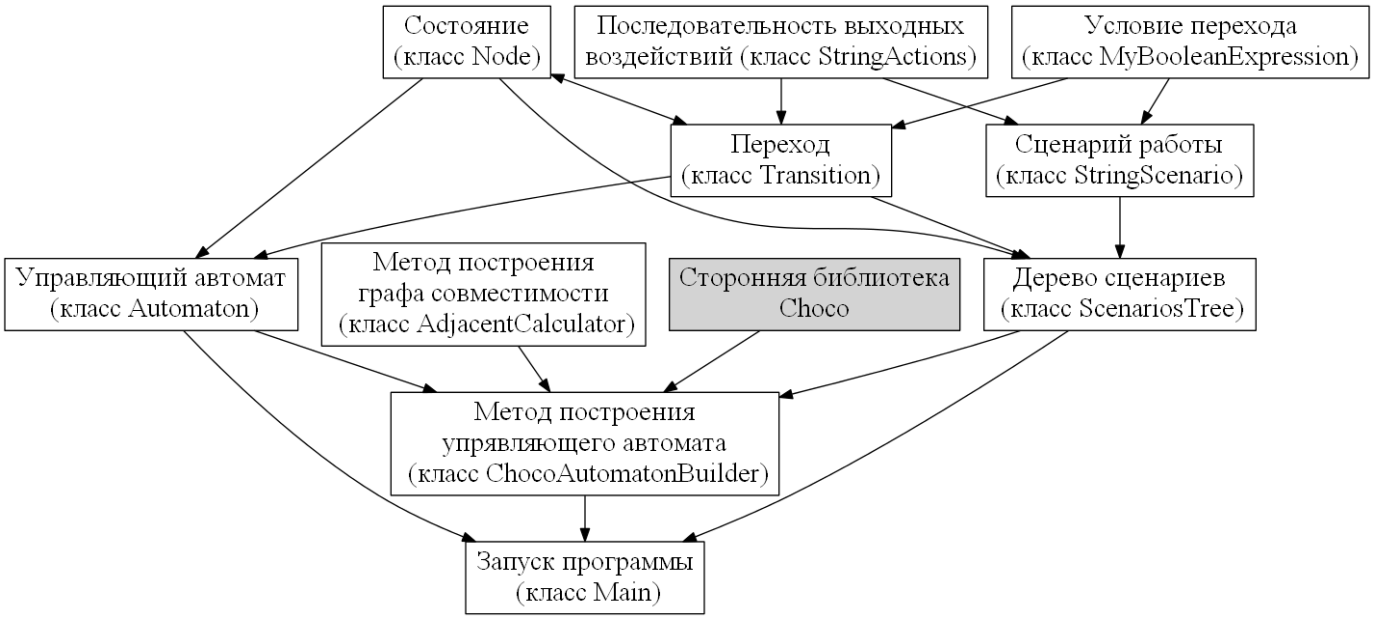

Рис. 15. Структура программной реализации метода

На схеме серым цветом выделена сторонняя библиотека *Choco*, остальные представленные сущности были реализованы в процессе работы. На часть приведенных классов было получено свидетельство о регистрации программы для ЭВМ: Ульянцев В.И., Царев Ф.Н. Программное средство для построения графа совместимости вершин дерева сценариев работы программы // Свидетельство о регистрации программы для ЭВМ. № 2012616462. Дата регистрации – 18.07.2012. В приложении Б содержатся копия полученного свидетельства.

Опишем программную реализацию классов, трудоемких с алгоритмической точки зрения.

3.2.1. 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. То есть, 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);
              }
            }
          }
        }
      }
    }
  }
```

3.2.2. Программная реализация этапа построения набора ограничений на целочисленные переменные

Построение набора ограничений на целочисленные переменные реализовано в классе `ChocoAutomatonBuilder`. Статический метод `build(…)` возвращает управляющий автомат типа `Automaton` и принимает на вход следующие параметры:

- дерево сценариев `tree` класса `ScenariosTree`;
- размер искомого автомата `size`;

- логическая переменная `isComplete`, соответствующая требованию полноты искомого автомата;
- `modelPrintWriter` – экземпляр класса `PrintWriter` для вывода модели построенных ограничений.

Данный метод создает модель ограничений `model` класса `CPModel`, создает и добавляет ограничения при помощи метода `model.addConstraints(…)` и строит искомый автомат при помощи внутреннего метода `buildAutomatonFromModel(…)`. Ограничения создаются следующим образом.

1. Ограничение $x_0 = 0$ на языке библиотеки описывается как `Choco.eq(nodesColorsVars[0], 0)`. В дальнейшем переменным x соответствует массив `nodesColorsVars` типа `IntegerVariable[]`.
2. Ограничения вида $(x_v = i) \Rightarrow (x_u = y_{i,e,f})$ создаются при помощи внутреннего метода `getTransitionsConstraints(size, tree, nodesColorsVars)`.
3. Ограничения вида $x_v \neq x_u$ создаются при помощи внутреннего метода `getAdjacentConstraints(tree, nodesColorsVars)`.
4. Ограничения видов $z_{i,e,f} = 1 \Leftrightarrow (x_{v_1} = i \vee \ldots \vee x_{v_n} = i)$ и $\left(\sum_{f \in F_e} (z_{i,e,f} \cdot c(f)) = 0\right) \vee \left(\sum_{f \in F_e} (z_{i,e,f} \cdot c(f)) = 2^m\right)$ создаются и добавляются при помощи метода `getCompleteConstraints(size, tree, nodesColorsVars)` только при условии истинности параметра `isComplete`.

4. ЭКСПЕРИМЕНТАЛЬНЫЕ ИССЛЕДОВАНИЯ

Экспериментальные исследования проводились по следующей схеме.

1. Генерация при помощи разработанного средства `automaton-generator.jar` случайного автомата A, удовлетворяющего требованию полноты, с заданным числом состояний, числом событий равным двум, числом входных переменных равным трем, числом выходных воздействий равным двум, процентом используемых переходов 50.
2. Генерация по автомату A сценариев работы при помощи разработанного средства `scenarios-generator.jar`. Каждый из сценариев работы является случайным путем в автомате A. Суммарная длина сценариев задается заранее, в среднем длина сценария (число переходов автомата) работы в два раза превышает число состояний автомата.
3. Запуск разработанного программного средства `builder.jar` на сгенерированных сценариях. Ищется автомат с заданным числом состояний, равным числу состояний A. При этом производится запуск средства для поиска автомата A_C, удовлетворяющего требованию полноты, и поиск автомата A_N без такого обязательства.
4. Проверка автоматов A_C и A_N на полноту и на изоморфизм исходному управляющему автомату A при помощи разработанных программных средств `completeness-checker.jar` и `isomorphism-checker.jar`. При этом автомат A_C в каждом исследовании удовлетворял требованию полноты, что говорит о корректности разработанного средства.
5. По аналогии с работой [31] проводилось сравнение поведений автоматов A_C и A_N с поведением автомата A с использованием дополнительных сценариев, не использовавшихся в обучающем наборе. По автомату A

генерировался дополнительный набор сценариев работы, состоящий из 1000 сценариев, длина каждого из которых в четыре раза больше числа состояний автомата *A*. После этого для каждого сценария проверялось, удовлетворяет ли он автоматам A_C и A_N (есть ли в них такой путь); записывались доли P_C и P_N удовлетворяющих сценариев работы.

Эксперименты проводились для числа состояний, равным 4, 6, 8 и 10; для суммарной длины сценариев работы, равной 800, 900, 1000, 1100 и 1200. Для каждого сочетания числа состояний и суммарной длины сценариев проводилось по 10 запусков. Исследования заняли около недели работы персонального компьютера с процессором *Intel Core2 Quad CPU* частотой 2.66 *GHz*. Приведем результаты проведенных экспериментов.

В табл. 2 приведено среднее время работы разработанного метода. При вычислении средних значений не брались в расчет запуски, на которых время построения превышало 30 секунд. Параметры таких запусков приведены в табл. 3 (так один из запусков закончился через 70 часов после начала). Причина такого продолжительного времени работы не выявлена, возможна доработка поисковой стратегии с использованием библиотеки *Choco*: использование нестандартной стратегии перезапуска, указание основных и вспомогательных переменных. Отметим, что задача *NP*-трудна, универсального способа быстрого решения за полиномиальное время может не существовать.

Таблица 2. Среднее время построения автоматов

Число состояний	Длина сценариев	Средний размер дерева сценариев	Среднее время построения A_N, сек.	Среднее время построения A_C, сек.
4	800	773.3	1.306	1.406
4	900	871.8	1.389	1.494
4	1000	968.0	1.836	1.947
4	1100	1068.9	2.131	2.225
4	1200	1174.3	2.809	2.864
6	800	732.7	1.886	2.011
6	900	840.1	2.045	2.382

6	1000	927.7	2.222	2.487
6	1100	1043.5	2.672	2.872
6	1200	1148.2	3.305	3.542
8	800	720.8	2.091	2.417
8	900	822.4	2.437	2.906
8	1000	885.8	2.939	3.089
8	1100	1010.1	3.946	4.199
8	1200	1114.5	4.464	5.433
10	800	657.5	2.455	3.075
10	900	781.6	3.445	4.555
10	1000	882.0	3.359	5.076
10	1100	965.2	4.639	6.303
10	1200	1072.1	5.206	5.950

Таблица 3. Запуски с продолжительным временем построения

Число состояний	Длина сценариев	Размер дерева сценариев	Время построения A_N, сек.	Время построения A_C, сек.
6	900	854	2.422	50.578
6	900	860	6.516	28989.094
6	1100	1057	3.031	104.235
8	900	777	58.062	81.719
8	1200	1138	115.531	305.578
8	1200	1142	11.625	745.672
8	1200	1105	5.203	30.375
10	800	627	2.141	4558.203
10	900	800	7.578	100705.619
10	900	807	2.984	169673.662
10	1000	822	3.015	132.797
10	1000	899	7.219	778.281
10	1000	909	6.250	809.984
10	1100	995	6.047	875.313
10	1100	1009	17.156	255519.828

В табл. 4 приведены показатели построенных автоматов. В третьей колонке приведены доли удовлетворяющих требованию полноты автоматов A_N. В четвертой и пятой колонках приведены усредненные доли автоматов A_N и A_C, которые изоморфны исходному автомату A. В шестой и седьмой колонках приведены усредненные значения коэффициентов «схожести поведения» P_N и P_C, соответственно. Выделенные ячейки соответствуют большему значению в сравнении двух запусков.

Таблица 4. Показатели построенных автоматов

Число состояний	Длина сценариев	Доля полных автоматов среди A_N	Доля изоморфных A_N автоматов	Доля изоморфных A_C автоматов	Среднее значение параметра P_N	Среднее значение параметра P_C
4	800	0.9	0.5	0.5	**98.7**	98.5
4	900	0.9	0.5	**0.6**	98.5	**100.0**
4	1000	1.0	0.6	0.6	99.6	**99.8**
4	1100	1.0	0.7	0.7	99.5	99.5
4	1200	1.0	0.8	0.8	100.0	100.0
6	800	1.0	0.3	0.3	100.0	100.0
6	900	0.7	0.3	0.3	94.8	**99.3**
6	1000	0.9	0.2	0.2	99.0	**100.0**
6	1100	0.9	0.4	0.4	98.7	**100.0**
6	1200	0.9	0.3	**0.4**	97.7	**99.0**
8	800	0.8	0.1	**0.2**	95.1	**97.6**
8	900	0.7	**0.5**	0.4	**93.1**	91.1
8	1000	0.8	0.4	**0.5**	93.5	**95.3**
8	1100	0.8	0.5	**0.6**	98.7	**100.0**
8	1200	0.5	0.1	**0.2**	94.3	**94.5**
10	800	0.4	0.1	0.1	79.0	**88.3**
10	900	0.6	0.2	0.2	77.1	**82.8**
10	1000	0.4	0.0	**0.2**	**87.8**	84.8
10	1100	0.4	0.2	**0.3**	**90.5**	89.8
10	1200	0.7	0.1	0.1	92.3	**93.8**

Приведенные значения позволяют говорить о следующем.

- Доля полных автоматов среди тех, которые были построены без обязательного удовлетворения требованию полноты, уменьшается с ростом числа состояний.
- Доля автоматов, изоморфных исходному, выше в среднем при построении автоматов, удовлетворяющих требованию полноты.
- В среднем, «поведение» на сценариях построенных автоматов не на много отличается от поведений исходных автоматов, но падает с ростом числа состояний. Существенной разницы среди усредненных значений «похожести» A_N и A_C на A не выявлено.

ЗАКЛЮЧЕНИЕ

Настоящая диссертация направлена на повышение уровня автоматизации производства программного обеспечения – предложен метод построения управляющих автоматов по сценариям работы программы, основанный на методах решения задачи удовлетворения ограничений.

В диссертации проведен аналитический обзор работ по построению автоматных моделей по обучающим примерам, обзор существующих методов решения задачи удовлетворения ограничений. Обзор показал, что методы удовлетворения ограничений ранее не применялись для построения автоматных моделей. Также был произведен выбор библиотеки *Choco* для решения задачи удовлетворения ограничений.

Была произведена разработка и реализация метода построения управляющих автоматов. Разработанный метод состоит из пяти этапов: построение дерева сценариев, построение графа совместимости, построение ограничений на целочисленные переменные, решение задачи удовлетворения построенных ограничений и, наконец, построение искомого автомата, в случае его существования. Предлагаемый подход оперирует с тремя типами целочисленных переменных, пятью типами ограничений на данные переменные. Разработка проводилась на языке программирования *Java*. Части разработанного средства зарегистрированы: получено свидетельство о регистрации программы для ЭВМ.

Качественное отличие предложенного метода от существующих заключается в предоставлении возможности построения управляющих автоматов, удовлетворяющих требованию полноты. Экспериментальные исследования показали, что метод показывает высокую производительность на различных задачах, справляясь с большинством задач менее, чем за минуту работы персонального компьютера. Также существуют задачи построения автоматов, удовлетворяющих требованию полноты, которые вычисляются несколько часов – построенные ограничения в данных случаях

непросто разрешить даже современным производительным методам решения задачи удовлетворения ограничений.

Дальнейшие исследования могут проводиться в следующих направлениях: применение методов удовлетворения ограничений для решения других задач построения управляющих автоматов по экспертным данным; исследование возможности построения автоматов по экспертным данным, содержащим ошибки, вещественным сценариям работы; построение автоматов с учетом заданных темпоральных свойств. Возможна доработка алгоритма построения дерева сценариев с использованием алгоритмов слияния состояний, доработка построения ограничений с применением техники добавления предикатов нарушения симметрии, доработка программного средства для повышения производительности разработанного метода.

ИСТОЧНИКИ

1. *Александров А. В., Казаков С. В., Сергушичев А. А., Царев Ф. Н., Шалыто А. А.* Применение эволюционного программирования на основе обучающих примеров для генерации конечных автоматов, управляющих объектами со сложным поведением // Известия РАН. Теория и системы управления. 2013. № 3, с. 85–100
2. *Гладков Л. А., Курейчик В. В., Курейчик В. М.* Генетические алгоритмы. М.: Физматлит. 2006.
3. *Егоров К. В., Царев Ф. Н., Шалыто А. А.* Применение генетического программирования для построения автоматов управления системами со сложным поведением на основе обучающих примеров и спецификации // Научно-технический вестник Санкт-Петербургского государственного университета информационных технологий, механики и оптики, № 5 (69), 2010, с. 81-86.
4. *Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К.* Алгоритмы. Построение и анализ. М.: Вильямс. 2010.
5. *Кларк Э., Грамберг О., Пелед Д.* Верификация моделей программ. М.: МЦНМО, 2002.
6. *Левенштейн В. И.* Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академии Наук СССР 163.4, с. 845–848.
7. *Николенко С. И., Тулупьев А. Л.* Самообучающиеся системы. М.: Издательство МЦНМО, 2009.
8. *Поликарпова Н.И., Точилин В.Н., Шалыто А.А.* Метод сокращенных таблиц для генерации автоматов с большим числом входных переменных на основе генетического программирования // Известия РАН. Теория и системы управления, 2010, № 2, с. 100–117
9. *Поликарпова Н. И., Шалыто А. А.* Автоматное программирование. СПб: Питер, 2009.
10. *Скиена С.* Алгоритмы. Руководство по разработке. СПб: БХВ-Петербург, 2011.

11. *Ульянцев В.И., Царев Ф.Н.* Применение методов решения задачи о выполнимости булевой формулы для построения управляющих конечных автоматов по сценариям работы // Научно-технический вестник информационных технологий, механики и оптики. 2012. №1(77), с. 96-100.
12. *Фридман А., Меннон П.* Теория и проектирование переключательных схем. М.: Мир, 1978.
13. *Хопкрофт Дж., Мотвани Р., Ульман Дж.* Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2002.
14. *Царев Ф. Н.* Диссертация на тему «Методы построения конечных автоматов на основе эволюционных алгоритмов». 2012.
15. *Царев Ф. Н.* Метод построения управляющих конечных автоматов на основе тестовых примеров с помощью генетического программирования // Информационно-управляющие системы. 2010. № 5, с. 31–36.
16. *Царев Ф. Н.* Совместное применение генетического программирования, конечных автоматов и искусственных нейронных сетей для построения системы управления беспилотным летательным аппаратом //Научно-технический вестник СПбГУ ИТМО. Выпуск 53. Автоматное программирование, с. 42–60.
17. *Belz A., Eskikaya B.* A Genetic Algorithm for Finite State Automata Induction with Application to Phonotactics / Proceedings of the ESSLLI-98 Workshop on Automated Acquisition of Syntax and Parsing, Saarbruecken, 1998, pp. 9–17.
18. *Buzhinsky I., Ulyantsev V., Shalyto A.* Test-Based Induction of Finite-State Machines with Continuous Output Actions / Preprints of the 2013 IFAC Conference on Manufacturing Modelling, Management, and Control, Saint Petersburg, 2013, pp. 1083-1088
19. *Chivilikhin D., Ulyantsev V.* Learning Finite-State Machines with Ant Colony Optimization // Lecture Notes in Computer Science, 2012, Volume 7461/2012,

pp. 268-275 http://rain.ifmo.ru/~ulyantsev/papers/2012/2012-ANTS12-Chivilikhin-Ulyantsev.pdf

20. Choco documentation. http://choco.svn.sourceforge.net/viewvc/choco/tags/choco-2.1.5/src/site/resources/tex/documentation/choco-doc.pdf
21. Choco, java library for constraint satisfaction problems (CSP) and constraint programming (CP). http://www.emn.fr/z-info/choco-solver/
22. *Cicchello O.*, *Kremer S.* Beyond EDSM. / Proceedings of the 6th International Colloquium Grammatical Inference, vol. 2484, pp. 37-48, 2002.
23. CSP 2008 Competition: available results. http://www.cril.univ-artois.fr/CPAI08/results/results.php?idev=15
24. CSP 2009 Competition: available results. http://www.cril.univ-artois.fr/CPAI09/results/results.php?idev=30
25. *Florêncio C.C.*, *Verwer. S.* Regular inference as vertex coloring / Algorithmic Learning Theory, pp. 81-95. Springer Berlin Heidelberg, 2012.
26. *Gold E. M.* Complexity of automaton identification from given data. // Information and Control, 1978, № 37, pp. 302−320.
27. *Hamming R.* Error detecting and error correcting codes. Bell System Technical Journal 29 (2), pp. 147–160.
28. *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 6339, Springer.
29. *Koza J.* Genetic programming. On the Programming of Computers by Means of Natural Selection. MA: The MIT Press, 1998.
30. *Lang K.* Faster algorithms for finding minimal consistent DFAs. Technical report, NEC Research Institute (1999)

31. *Lang K., Pearlmutter B., Price R.* Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm / Grammatical Inference. 1998. Lecture Notes in Computer Science. Vol. 1433, pp. 1 – 12.
32. *Lang K.* Random DFA's can be approximately learned from sparse uniform examples / Proceedings of the Fifth ACM Workshop on Computational Learning Theory, (New York, N.Y.), pp. 45-52, ACM, 1992.
33. Learning DFS from Noisy Samples. A contest from GECCO 2004. http://cswww.essex.ac.uk/staff/sml/gecco/NoisyDFA.html
34. *Lucas S., Reynolds J.* Learning DFA: Evolution versus Evidence Driven State Merging // The 2003 Congress on Evolutionary Computation (CEC '03). Vol. 1, pp. 351–358.
35. *Lucas S., Reynolds J.* Learning Deterministic Finite Automata with a Smart State Labeling Algorithm // IEEE Transactions on Pattern Analysis and Machine Intelligence. Vol. 27, №7, 2005, pp. 1063–1074.
36. *Lucas S.* Evolving Finite-State Transducers: Some Initial Explorations. Lecture Notes in Computer Science. Springer Berlin / Heidelberg. Volume 2610/2003, pp. 241–257. http://www.springerlink.com/content/41a34vg70fp1hltb/
37. *Metodi A., Codish M.* Compiling Finite Domain Constraints to SAT with BEE. Theory and Practice of Logic Programming, 2012.
38. Mistral. http://www.cril.univ-artois.fr/CPAI06/descriptionSolvers/Mistral.pdf
39. *Mitchell M., Holland J., Forrest S.* When will a genetic algorithm outperform hill climbing? / Advances in Neural Information Processing Systems 6, pp. 51–58. Morgan Kaufman, San Mateo, California, 1994.
40. *Oncina J., Garcia P.* Inferring Regular Languages in Polynomial Updated Time // Pattern Recognition and Image Analysys. Perez de la Blanca, Sanfeliu and Vidal (Eds.) World Scientific, 1992.
41. Presentation of SAT-Race results at the SAT’10 conference. http://baldur.iti.uka.de/sat-race-2010/downloads/SAT-Race-2010-Presentation.pdf

42. Sat4j, the Boolean satisfaction and optimization library in Java. http://www.sat4j.org/index.php
43. Sugar: A SAT-based Constraint Solver. http://bach.istc.kobe-u.ac.jp/sugar/
44. *Tomita M*. Dynamic construction of finite automata from examples using hill climbing // Proceedings of the 4th Annual Cognitive Science Conference (USA, 1982), pp. 105–108.
45. *Trakhtenbrot B., Barzdin Y*. Finite Automata: Behavior and Synthesis. North Holland Publishing Company, 1973.

ПРИЛОЖЕНИЕ А. ПРИМЕР НАБОРА ОГРАНИЧЕНИЙ

```
==== VARIABLES ====
Color_0 [0, 1]
Color_5 [0, 1]
T[~x]_0 [0, 1]
T[~x]_1 [0, 1]
Color_1 [0, 1]
T[x]_0 [0, 1]
T[x]_1 [0, 1]
Color_2 [0, 1]
Color_9 [0, 1]
Color_3 [0, 1]
Color_4 [0, 1]
Color_6 [0, 1]
Color_7 [0, 1]
Color_11 [0, 1]
Color_8 [0, 1]
Color_10 [0, 1]
used_T_~x_0 [0, 1]
used_T_~x_1 [0, 1]
used_T_x_0 [0, 1]
used_T_x_1 [0, 1]
0
1
==== MULTIPLE VARIABLES ====

==== CONSTRAINTS ====
eq ( { Color_0 [0, 1], 0 } )
implies ( { eq ( { Color_0 [0, 1], 0 } ), eq ( { Color_5 [0, 1],
T[~x]_0 [0, 1] } ) } )
```

```
     implies ( { eq ( { Color_0 [0, 1], 1 } ), eq ( { Color_5 [0, 1],
T[~x]_1 [0, 1] } ) } )
     implies ( { eq ( { Color_0 [0, 1], 0 } ), eq ( { Color_1 [0, 1],
T[x]_0 [0, 1] } ) } )
     implies ( { eq ( { Color_0 [0, 1], 1 } ), eq ( { Color_1 [0, 1],
T[x]_1 [0, 1] } ) } )
     implies ( { eq ( { Color_1 [0, 1], 0 } ), eq ( { Color_2 [0, 1],
T[~x]_0 [0, 1] } ) } )
     implies ( { eq ( { Color_1 [0, 1], 1 } ), eq ( { Color_2 [0, 1],
T[~x]_1 [0, 1] } ) } )
     implies ( { eq ( { Color_1 [0, 1], 0 } ), eq ( { Color_9 [0, 1],
T[x]_0 [0, 1] } ) } )
     implies ( { eq ( { Color_1 [0, 1], 1 } ), eq ( { Color_9 [0, 1],
T[x]_1 [0, 1] } ) } )
     implies ( { eq ( { Color_2 [0, 1], 0 } ), eq ( { Color_3 [0, 1],
T[~x]_0 [0, 1] } ) } )
     implies ( { eq ( { Color_2 [0, 1], 1 } ), eq ( { Color_3 [0, 1],
T[~x]_1 [0, 1] } ) } )
     implies ( { eq ( { Color_3 [0, 1], 0 } ), eq ( { Color_4 [0, 1],
T[x]_0 [0, 1] } ) } )
     implies ( { eq ( { Color_3 [0, 1], 1 } ), eq ( { Color_4 [0, 1],
T[x]_1 [0, 1] } ) } )
     implies ( { eq ( { Color_5 [0, 1], 0 } ), eq ( { Color_6 [0, 1],
T[~x]_0 [0, 1] } ) } )
     implies ( { eq ( { Color_5 [0, 1], 1 } ), eq ( { Color_6 [0, 1],
T[~x]_1 [0, 1] } ) } )
     implies ( { eq ( { Color_6 [0, 1], 0 } ), eq ( { Color_7 [0, 1],
T[x]_0 [0, 1] } ) } )
     implies ( { eq ( { Color_6 [0, 1], 1 } ), eq ( { Color_7 [0, 1],
T[x]_1 [0, 1] } ) } )
     implies ( { eq ( { Color_7 [0, 1], 0 } ), eq ( { Color_11 [0, 1],
T[~x]_0 [0, 1] } ) } )
```

```
    implies ( { eq ( { Color_7 [0, 1], 1 } ), eq ( { Color_11 [0, 1],
T[~x]_1 [0, 1] } ) } )
    implies ( { eq ( { Color_7 [0, 1], 0 } ), eq ( { Color_8 [0, 1],
T[x]_0 [0, 1] } ) } )
    implies ( { eq ( { Color_7 [0, 1], 1 } ), eq ( { Color_8 [0, 1],
T[x]_1 [0, 1] } ) } )
    implies ( { eq ( { Color_9 [0, 1], 0 } ), eq ( { Color_10 [0, 1],
T[~x]_0 [0, 1] } ) } )
    implies ( { eq ( { Color_9 [0, 1], 1 } ), eq ( { Color_10 [0, 1],
T[~x]_1 [0, 1] } ) } )
    neq ( { Color_2 [0, 1], Color_1 [0, 1] } )
    neq ( { Color_2 [0, 1], Color_0 [0, 1] } )
    neq ( { Color_5 [0, 1], Color_1 [0, 1] } )
    neq ( { Color_5 [0, 1], Color_0 [0, 1] } )
    neq ( { Color_7 [0, 1], Color_5 [0, 1] } )
    neq ( { Color_7 [0, 1], Color_2 [0, 1] } )
    neq ( { Color_9 [0, 1], Color_5 [0, 1] } )
    neq ( { Color_9 [0, 1], Color_2 [0, 1] } )
    ifonlyif ( { or ( { eq ( { Color_0 [0, 1], 0 } ), eq ( { Color_1 [0,
1], 0 } ), eq ( { Color_2 [0, 1], 0 } ), eq ( { Color_5 [0, 1], 0 } ), eq
( { Color_7 [0, 1], 0 } ), eq ( { Color_9 [0, 1], 0 } ) } ) , eq ( {
used_T_~x_0 [0, 1], 1 } ) } )
    ifonlyif ( { or ( { eq ( { Color_0 [0, 1], 1 } ), eq ( { Color_1 [0,
1], 1 } ), eq ( { Color_2 [0, 1], 1 } ), eq ( { Color_5 [0, 1], 1 } ), eq
( { Color_7 [0, 1], 1 } ), eq ( { Color_9 [0, 1], 1 } ) } ) , eq ( {
used_T_~x_1 [0, 1], 1 } ) } )
    ifonlyif ( { or ( { eq ( { Color_0 [0, 1], 0 } ), eq ( { Color_1 [0,
1], 0 } ), eq ( { Color_3 [0, 1], 0 } ), eq ( { Color_6 [0, 1], 0 } ), eq
( { Color_7 [0, 1], 0 } ) } ) , eq ( { used_T_x_0 [0, 1], 1 } ) } )
    ifonlyif ( { or ( { eq ( { Color_0 [0, 1], 1 } ), eq ( { Color_1 [0,
1], 1 } ), eq ( { Color_3 [0, 1], 1 } ), eq ( { Color_6 [0, 1], 1 } ), eq
( { Color_7 [0, 1], 1 } ) } ) , eq ( { used_T_x_1 [0, 1], 1 } ) } )
```

```
    or ( { regular ( { used_T_~x_0 [0, 1], used_T_x_0 [0, 1] } ),
regular ( { used_T_~x_0 [0, 1], used_T_x_0 [0, 1] } ) } )
    or ( { regular ( { used_T_~x_1 [0, 1], used_T_x_1 [0, 1] } ),
regular ( { used_T_~x_1 [0, 1], used_T_x_1 [0, 1] } ) } )
```

ПРИЛОЖЕНИЕ Б. СВИДЕТЕЛЬСТВО О РЕГИСТРАЦИИ ПРОГРАММЫ ДЛЯ ЭВМ

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2012616462

Программное средство для построения графа совместимости вершин дерева сценариев работы программы

Правообладатель(ли): *федеральное государственное бюджетное образовательное учреждение высшего профессионального образования «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики» (RU)*

Автор(ы): *Ульянцев Владимир Игоревич, Царев Федор Николаевич (RU)*

Заявка № 2012614587

Дата поступления 5 июня 2012 г.

Зарегистрировано в Реестре программ для ЭВМ *18 июля 2012 г.*

Руководитель Федеральной службы по интеллектуальной собственности

Б.П. Симонов