МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное агентство по образованию
Государственное образовательное учреждение высшего профессионального образования

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»

Факультет **Информационных технологий и программирования**

Направление **Прикладная математика и информатика**
Специализация : **Технологии программирования**
Академическая степень **магистр прикладной математики и информатики**

Кафедра **Компьютерных технологий** Группа **6538**

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

Совместное применение генетического программирования и верификации моделей для построения автоматов управления системами со сложным поведением

Автор магистерской диссертации Егоров К.В. (подпись)
(Фамилия, И., О.)

Научный руководитель Шалыто А.А. (подпись)
(Фамилия, И., О.)

Руководитель магистерской программы (подпись)
(Фамилия, И., О.)

К з а щ и т е д о п у с т и т ь
Зав. кафедрой ВАСИЛЬЕВ В.Н. (подпись)
(Фамилия, И., О.)

“____”_____20 ____г.

Санкт-Петербург, 2010 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Автоматное программирование – это парадигма программирования, в рамках которой программы предлагается проектировать в виде совокупности взаимодействующих автоматизированных объектов управления [1]. В автоматных программах выделяют три типа объектов: поставщики событий, система управления и объекты управления. Система управления представляет собой конечный автомат или систему взаимодействующих конечных автоматов. Поставщики событий генерируют события, а система управления по каждому событию может совершать переход, считывая значения входных переменных у объектов управления для проверки условия перехода.

Для многих задач автоматы удается строить эвристически, однако существуют задачи, для которых такое построение затруднительно [2–4]. В рамках работы [5] был предложен подход к построению управляющих конечных автоматов на основе обучающих примеров. При использовании такого метода на начальном этапе проектирования автомата (модели) выделяются события (*e1, e2,* …), входные переменные (*x1, x2,* …) и выходные воздействия (*z1, z2*, …). В качестве тестов для управляющего конечного автомата рассматривались пары последовательностей, одна из которых описывает события и входные переменные, поступающие на вход автомату, а вторая – выходные воздействия, которые должен вырабатывать автомат при обработке этих событий.

Предложенный подход обладает тем недостатком, что при построении неправильного (ошибочного) автомата пользователю приходится снова и снова модифицировать тесты или добавлять новые, пока не будет построен требуемый конечный автомат. Такие действия могут занять много времени, так как построение сложного автомата генетическими алгоритмами требует рассмотрения определенного числа поколений. В итоге, может оказаться, что построение вручную займет меньше времени, чем использование генетического алгоритма.

В любом случае, даже построив кажущийся правильным и проходящий все тесты конечный автомат, нельзя гарантировать его поведение при других входных воздействиях. Под правильностью подразумевается корректное поведение построенного автомата при любых возможных вариантах входных событий и входных переменных. Так как вариантов последовательностей входных событий бесконечно много, а тесты описывают только конечное число вариантов поведения автомата, то нельзя говорить о какой бы то ни было корректности построенного автомата.

Но какие должны быть выходные воздействия при поступлении на вход автомату той последовательности событий, которая не была описана в тестах? Ответ на данный вопрос можно дать двумя способами. Первый, если автомат ведет себя неправильно при определенной последовательности входных событий и переменных, то добавим новый тест и построим автомат заново. Второй вариант, использовать верификацию модели при построении конечных автоматов.

Первый вариант плох тем, что приходится вручную искать ошибки в построенном автомате, и тем, что мы и вовсе можем не заметить ошибку. Такие ошибки могут возникнуть при эксплуатации системы, исправление которых может стать слишком дорогим, а в некоторых случаях, например при проектировании систем жизнеобеспечения, ошибки просто не допустимы и все равно требуется формальное доказательство корректности модели (программы или системы).

Второй вариант подхода к построению конечных автоматов позволяет описывать бесконечное число вариантов поведения автомата. Такой подход позволяет запрещать какие-нибудь варианты развития событий, позволяет делать утверждения не только о конкретной последовательности входных и выходных воздействий, но и накладывать на них определенные условия и ограничения, утверждать о событиях в будущем.

Далее в настоящей работе будет дан краткий обзор методов верификации, определены основные понятия и объяснено, как верификация позволяет делать такие утверждения об автоматной программе.

ГЛАВА 1. ВЕРИФИКАЦИЯ АВТОМАТНЫХ ПРОГРАММ

Метод проверки того, что программная система соответствует заявленной спецификации (обладает необходимыми свойствами или удовлетворяет определенным требованиям (утверждениям)), называется *верификацией*. К сожалению, верифицировать систему обычно намного сложнее, чем ее создать. Это также является одним из факторов того, что использование верификации в процессе создания самих автоматов позволит не только генерировать автоматы с заранее заданным поведением, но и в какой-то степени избавляет нас от необходимости верифицировать систему после окончания ее построения. При этом отметим, что аккуратное и точное описание свойств автомата на языке верификатора также является непростой задачей и требует определенных навыков и умений обращения с темпоральными свойствами.

Наиболее практичным в настоящее время является метод верификации, называемый *Model Checking* [6, 7]. При его использовании процесс верификации состоит из трех этапов. Первый из них, моделирование программы состоит в преобразовании программы в формальную модель с конечным числом состояний для последующей верификации. Второй этап, спецификация – формальная запись утверждений, которые требуется проверить. На третьем этапе, выполняется собственно верификация – алгоритмическая проверка выполнения спецификации для модели.

Сложность такого подхода заключается в том, что после построения модели и ее верификации, необходимо обратное преобразование ошибки в модели в ошибку в программе. Причем не всегда корректность модели означает соответствие программы спецификации, так как при построении модели мы переходим на другой уровень абстракции, теряя определенные данные и связи в программе. Данный процесс представлен на рис. 1.

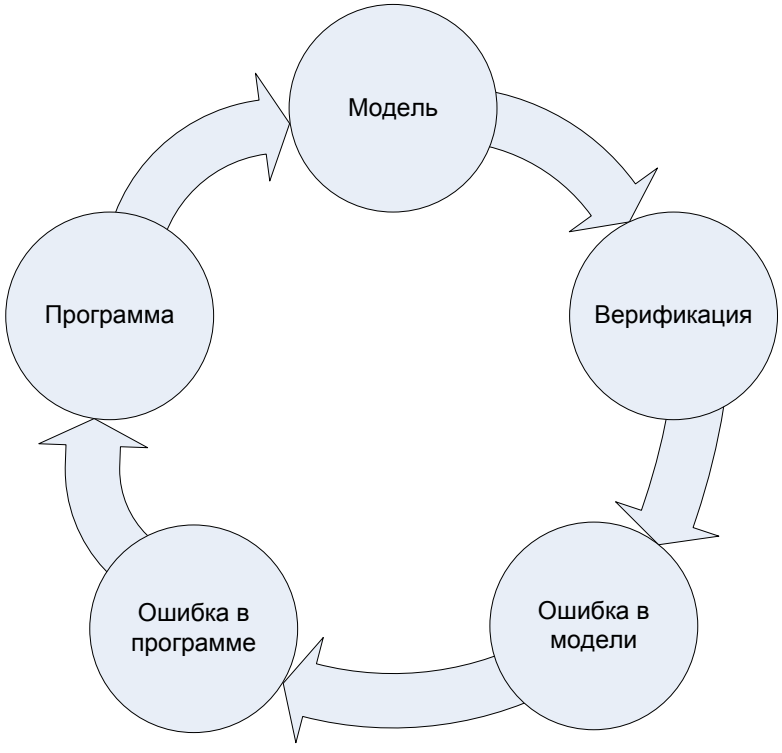


Рис. 1. Стандартный процесс верификации программы

Однако, метод *Model Checking* достаточно хорошо подходит для случая автоматных программ, так как нет необходимости преобразовывать программу в модель, и после верификации, в случае обнаружения контрпримера, совершать обратное преобразование. Это объясняется тем, что автомат уже является моделью, пригодной для верификации, и не требует никаких дополнительных преобразований и упрощений. Такая особенность автоматов позволяет строить утверждения о программе в терминах автоматов, что упрощает

их верификацию по сравнению с программами, написанными традиционным путем (без явного выделения состояний).

В предлагаемом методе будет проводиться верификация не всей автоматной программы, а только ее модель (конечный автомат). Это немного упрощает задачу, но так же, как и при создании автоматной программы на основе тестов, мы считаем поставщиков событий и объекты управления достаточно простыми, а вся сложная логика вынесена в автоматную модель. При верификации будем рассматривать поставщиков событий и объекты управления в качестве «внешней среды», которая ничего не помнит о последовательности переходов рассматриваемого автомата. Таким образом, в любой момент времени может быть получено любое событие, и любое условие на переходе может быть как истинным, так и ложным. Это приводит к тому, что автомат может совершить любой переход из данного состояния. Такой подход уже был рассмотрен в работе [8]. Но это не является упрощением модели в случае вынесения всей логики в автоматную.

В настоящей работе требования к программе формулируются в виде формул темпоральной логики линейного времени (*Linear Temporal Logic*, *LTL*). Далее кратко опишем синтаксис и семантику этого языка. Сразу заметим, что как выбор языка темпоральной логики, так и выбор верификатора не влияет на предложенный метод построения автоматных программ генетическими алгоритмами. Это позволяет применять его с другими программными средствами, реализующими верификацию методом *Model Checking*.

1.1. Язык логики линейного времени

Как уже отмечалось ранее, для описания поведения автомата будем применять утверждения, написанные на языке *LTL*. Синтаксис *LTL* включает в себя пропозициональные переменные *Prop*, булевы связки ($\neg$, Λ, V) и темпоральные операторы. Последние применяются для составления утверждений о событиях в будущем и интерпретируются как $I: Prop \rightarrow \{True, False\}$.

Логика линейного времени расширяет классическую логику, добавляя временные операторы. В нем время линейно и дискретно, и в каждый момент времени любая пропозициональная переменная может быть истиной или ложной.

Будем использовать следующие темпоральные операторы:

- **X** (ne**X**t) – «*Xp*» – в следующий момент выполнено p;
- **F** (in the **F**uture) – «*Fp*» – в некоторый момент в будущем будет выполнено p;
- **G** (**G**lobally in the future) – «*Gp*» – всегда в будущем выполняется p;
- **U** (**U**ntil) – «*pUq*» – существует состояние, в котором выполнено q и во всех предыдущих выполняется p;
- **R** (**R**elease) – «*pRq*» – либо во всех состояниях выполняется q, либо существует состояние, в котором выполняется p, а во всех предыдущих выполнено q.

Множество *LTL* формул таково:

- пропозициональные переменные *Prop*;
- *True*, *False*;
- φ и ψ – формулы, то
  - $\neg\varphi$, $\varphi\Lambda\psi$, $\varphi V\psi$ – формулы;
  - $X\varphi$, $F\varphi$, $G\varphi$, $\varphi U\psi$, $\varphi R\psi$ – формулы.

Логика линейного времени говорит о всех путях. Таким образом, логика *LTL* предполагает, что некоторое утверждение будет выполняться для всех путей. Поэтому можно строить доказательство от противного и проверять существование пути, на котором будет выполняться отрицание данной формулы. Если такой путь не будет найден, то формула выполнима.

1.2. Алгоритм верификации

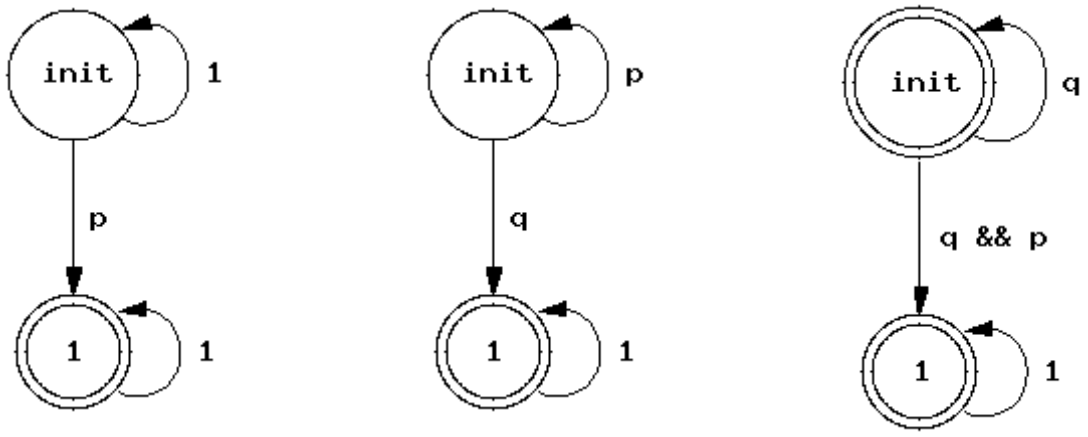
Алгоритм верификации основан на том, что как модель автоматной программы, так и *LTL*-формулу можно представить в виде автомата *Бюхи*. Формально он определяется пятеркой (S, E, T, s_0, F), где

- S – конечное множество состояний;
- E – множество меток переходов;
- $T \subseteq S \times E \times S$ – множество переходов;
- s_0 – начальное состояние;
- $F \subseteq S$ – множество допускающих состояний.

Тогда путь в этом графе $\pi = s_0, s_1, s_2, \ldots s_n, \ldots$, для которого выполнено $T(s_{i-1}, e, s_i)$, где e – метка перехода, будет последовательностью вычислений системы. Путь является допускающим, если существует состояние из множества F, встречающееся бесконечно часто.

Подробно о трансляции *LTL*-формулы в автомат Бюхи изложено в работах [7, 9, 10].

Приведем несколько примеров автоматов Бюхи, построенного по основным формулам. На рис. 2 приведены автоматы, построенные для темпоральных операторов **F**uture, **U**ntil и **R**elease. Как «init» обозначены начальные состояния, а состояния, отмеченные двойным кругом, являются допускающими.



а б в

Рис. 2. Автоматы Бюхи, построенные для LTL-формул Fp (а), pUq (б), pRq (в)

Модель автоматной программы представляет собой автомат Бюхи, в котором метка на переходе – это выполнимость определенного предиката. Под предикатом будем понимать утверждение о текущем переходе, например, вызванные автоматом действия в объектах управления или состояние, в которое перешел автомат.

Для доказательства выполнимости некоторой *LTL*-формулы на автомате Бюхи будем проверять, что пересечение верифицируемого автомата Бюхи и автомата Бюхи, соответствующего отрицанию *LTL*-формулы, пусто. Для этого требуется доказать, что язык автомата пересечения пуст. Из сказанного следует, что алгоритм верификации может быть следующим: строится автомат Бюхи для верифицируемой автоматной программы, по отрицанию *LTL*-формулы строится автомат Бюхи, затем строится автомат пересечения, а после этого проверяется, что этот автомат не допускает ни одного слова.

В связи с тем, что рассматриваются бесконечные слова, то, как доказано в работе [7], для пустоты пересечения достаточно доказать, что ни одно допускающее состояние не принадлежит сильной компоненте связанности, которая достижима из начального состояния (не существует цикла, проходящего через допускающее состояние). Таким образом, при

нахождении цикла, достижимого из начального состояния, будет построен контрпример – путь в модели, на котором не выполняется *LTL*-формула.

При верификации обычно применяют двойной обход в глубину [7], преимущество которого состоит в том, что для реализации этого алгоритма не требуется построение автомата-пересечения целиком – можно строить состояния пересечения автоматов по мере их достижения. Это дает выигрыш на больших моделях.

Общая идея алгоритма такова: обходим в глубину автомат пересечения, при достижении допускающего состояния для проверки достижимости самого себя запускаем второй обход в глубину из данного состояния. Если оказалось, что допускающее состояние достижимо из самого себя, то цикл найден. Следовательно, исходная *LTL*-формула не выполняется на автомате Бюхи, представляющем модель программы, и найден контрпример.

Приведем рекурсивный алгоритм двойного обхода в глубину на псевдокоде из работы [7].

```
procedure emptiness
    for all q0 ϵ Q0 do
        dfs1(q0);
    terminate(False);
end procedure

procedure dfs1(q)
    local q′;
    hash(q);
    for all последователей q′ вершины q do
        if q′ не содержится в хэш-таблице then dfs1(q′);
    if accept(q) then dfs2(q);
end procedure

procedure dfs2(q)
    local q′;
    flag(q);
    for all последователей q′ вершины q do
        if q′ в стеке dfs1 then terminate(True);
        else if q′ не является помеченной then dfs2(q′);
        end if;
end procedure
```

Приведенный алгоритм работает следующим образом: когда первый обход в глубину (*Depth-first search, DFS*) покидает состояние, он вызывает второй *DFS* для обнаружения циклов. Если второй *DFS* пришел в состояние, содержащееся в стеке первого *DFS*, то цикл найден. Тогда стек первого *DFS* содержит конечный префикс контрпримера, а второй обнаружил цикл, который служит бесконечным суффиксом. Таким образом, язык пересечения автоматов Бюхи не пуст.

1.3. Программная реализация верификатора

В настоящей работе использовался верификатор, реализованный в работе [11]. Верификатор на вход получает модель автоматной программы и *LTL*-формулу. После проверки модели верификатор либо сообщает, что формула выполняется, либо приводит контрпример в виде последовательности состояний и переходов в конечном автомате (рис. 3).

Рис. 3. Схема работы верификатора

Как уже отмечалось выше, выбор верификатора и языка темпоральной логики не имеет значения, но использовалась именно эта реализация, так как этот верификатор написан на языке программирования *Java* и предназначен для проверки утверждений об автоматных программах. Верификатор предоставляет набор классов для трансляции *LTL*-формул в автомат Бюхи и дальнейшей проверки пустоты языка пересечения отрицания *LTL*-формулы и языка, допускаемого автоматом модели.

Верификатор из работы [11] реализует алгоритм двойного обхода в глубину, описанный выше. При этом, в случае обнаружения ошибки в модели и существования контрпримера, опровергающего утверждение о ней, данному алгоритму не требуется построение полного автомата пересечения двух автоматов Бюхи. Однако, в случае выполнимости формулы, полный автомат пересечения все-таки будет построен.

Верификатор позволяет проверять утверждения о вызванных действиях, их последовательности, событиях и аналогичные предикаты. Указанное средство позволяет формулировать и верифицировать следующие предикаты:

- *wasEvent(e)* – переход совершен по событию *e*;
- *isInState(s)* – переход совершен в состояние *s*;
- *wasInState(s)* – переход совершен из состояния *s*;
- *wasAction(z)* – во время перехода было вызвано действие *z*;
- *wasFirstAction(z)* – во время перехода первым вызванным действием было *z*.

Однако в ряде случаев выразительности таких предикатов может не хватать для проверки утверждений, которые могут потребоваться. Поэтому используемый верификатор дает возможность создавать собственные предикаты. Это позволяет не строить «хитрые» утверждения, использующие стандартные предикаты и логические операторы. Например, для проверки утверждения «*действие o1.z2 вызывается через одно после o1.z1*» достаточно написать один метод на языке *Java*, которому доступна информация о совершенном переходе. Таким образом, можно легко проверять такого рода утверждения, не прибегая к сложной комбинации предопределенных предикатов.

При этом утверждения об автоматной программе записываются обычной строкой, например, «*G(wasEvent(p1.e1))*» или «*F(wasAction(o1.z1))*». Таким образом, синтаксис *LTL*-формул остался таким же, как в разд. 1.1. Это позволяет записывать утверждения о программе человеку, который знает только синтаксис и семантику языка *LTL*.

Выводы по главе 1

В настоящем разделы было дано понятие верификации, описан язык логики линейного времени (*LTL*), дано описание и основная идея алгоритма верификации. Предлагаемый метод может использовать любой верификатор, позволяющий проверять утверждения об автоматных программах, но в работе использовался верификатор из работы [11].

Далее будет описан предлагаемый метод построения автоматных программ генетическим алгоритмом на основе тестов и темпоральных свойств. Темпоральные свойства будут записываться на языке *LTL* и верифицироваться верификатором, описанным в настоящем разделе.

ГЛАВА 2. ОПИСАНИЕ ПРЕДЛАГАЕМОГО МЕТОДА

Исходными данными для построения конечного автомата управления системой со сложным поведением являются:

- список событий;
- список входных переменных;
- список выходных воздействий;
- набор тестов, каждый из которых содержит последовательность Input[*i*] событий, поступающих на вход конечному автомату, и соответствующую ей эталонную последовательность Answer[*i*] выходных воздействий;
- набор темпоральных свойств, записанных на языке логики *LTL*.

Отметим, что метод, описанный в этом разделе, основан на методе из построения конечных автоматов на основе обучающих примеров из работы [5], однако предлагаемый подход использует результат верификации в генетическом алгоритме на стадии вычисления функции приспособленности и на стадии мутации. Такая интеграция верификации в уже разработанный метод автоматического создания программ на основе только тестов, по сути, позволяет использовать любое средство для верификации модели с любым входным языком.

Так же, как при создании автоматной модели только на основе тестов, запись *LTL*-формул не предполагает реализации объектов управления и поставщиков событий заранее – они могут быть созданы и после создания модели. Однако, можно создавать модель по уже готовой реализации поставщиков событий и объектов управления. Первый случай может возникнуть, когда мы создаем программу с нуля и предоставляем только интерфейс поставщиков событий и объектов управления, откладывая их реализацию. Второй вариант – когда у нас уже есть программа с неправильной моделью, или реализация этих объектов заранее известна.

Стоит отметить, что заранее мы знаем только входные воздействия, входные условия и выходные воздействия, то есть мы можем строить утверждения только о них. Это означает, что мы не можем использовать в *LTL*-формулах предикаты о состояниях, так как мы не знаем заранее ничего о структуре конечного автомата. С другой стороны, это можно считать и преимуществом, так как мы заранее не ограничиваем себя априорными знаниями о состояниях будущего автомата. Тем более, создавая программу таким образом (на основе тестов и темпоральных свойств), мы не знаем какая модель должна получиться, а только можем заранее описать ее требуемое поведение. Ведь иначе можно было бы создать ее вручную, а только затем верифицировать.

2.1. Представление конечного автомата в виде хромосомы генетического алгоритма

В настоящей работе используется такое же представление конечных автоматов в виде хромосом генетического алгоритма, как и в работе [5].

Конечный автомат в алгоритме генетического программирования представляется в виде объекта, который содержит описания переходов для каждого из состояний и номер начального состояния. Для каждого из состояний хранится список переходов. Каждый переход описывается событием, при поступлении которого этот переход выполняется, и числом выходных воздействий, которые должны быть сгенерированы при выборе этого перехода.

Таким образом, в особи кодируется только «скелет» (рис. 4) управляющего конечного автомата, а конкретные выходные воздействия, вырабатываемые на переходах, определяются с помощью алгоритма расстановки пометок.

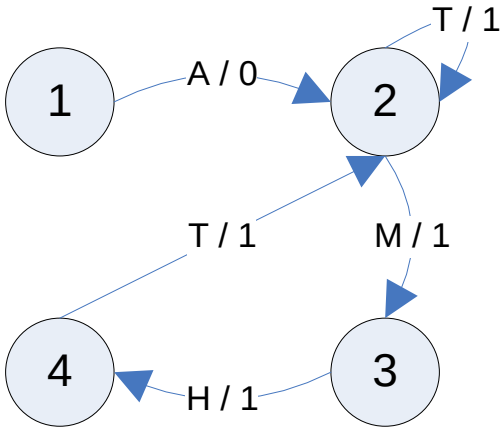


Рис. 4. Некоторые переходы «скелета» автомата

В настоящей работе генетический алгоритм остался таким же, как и в работе [5], то есть он имеет все те же стадии: создание начальной популяции, вычисление функции приспособленности, скрещивание, мутация. Также сохранилась стратегия эллитизма. Однако предлагается учитывать результат верификации при вычислении функции приспособленности и мутации, о чем будет написано ниже. Для начала отметим отличие в обработки входных переменных при генерации на основе тестов и при верификации.

2.1.1. Обработка входных переменных

Каждый переход может иметь условие, при котором он совершается. Такое условие записывается в виде логической формулы, которая задает ограничения на значения входных переменных. Например, можно записывать «*T [!x1 && x2]*», что означает, что переход совершится по событию *T* при условии невыполнимости *x1* и выполнимости *x2*. При создании автоматной модели только на основе тестов, такой переход считается отличным от перехода просто по событию *T* без всяких условий – можно считать, что это два разных события.

Однако, оба перехода идентичны для предиката *wasEvent(T)* («*переход совершен по событию T*»). При верификации, если переход был совершен по событию *T* или этому же событию, но с некоторыми условиями, то в обоих случаях данный предикат, естественно, будет выполнен. Заметим, что можно вводить предикаты и на условия на переходах, тогда такие переходы будут отличаться и для верификатора. Например, можно добавить предикат *wasTrue(x1)* (условие x1 верно), тогда он не будет выполняться на переходе «*T [!x1 && x2]*», но будет верен для перехода «*T [x1 && x2]*».

2.2. Вычисление функции приспособленности

В любом генетическом алгоритме ключевую роль играет вычисление функции приспособленности. Она позволяет количественно оценивать особи: чем больше функция приспособленности, тем лучше особь. Таким образом, независимо от выбора стратегии генетического алгоритма, функция приспособленности лучшей особи в популяции, как правило, должна расти. В любом случае поиск автоматной модели считается завершенным, когда найдена особь, для которой значение функции приспособленности превышает некоторое целевой значение, заданное заранее.

Особь, проходящая все тесты и удовлетворяющая всем темпоральным свойствам, является той, которую мы ищем. Это означает, что ее функция приспособленности должна быть больше, чем у особи, не проходящей определенное число тестов или не удовлетворяющей неким формулам.

Сразу заметим, что, так же как и в работе [5], в функции приспособленности должно учитываться общее число переходов в конечном автомате, однако вклад числа переходов

должен быть меньше, чем формул или тестов. Ведь нам важнее найти «правильную» модель, а не модель с наименьшим числом состояний.

В настоящей работе функция приспособленности является суммой трех частей. Каждая часть представляет собой вклад одного из критериев: успешность прохождения тестов, выполнимость *LTL*-формул, число переходов в конечном автомате.

Напомним, что функция приспособленности для тестов основана на редакционном расстоянии (расстоянии Левенштейна) [12]. Для ее вычисления выполняются следующие действия: на вход автомату подается каждая из последовательностей Input[i]. Обозначим последовательность выходных воздействий, которую сгенерировал автомат на входе Input[i] как Output[i]. После этого вычисляется величина $\mathrm{FF_1} = \frac{\sum_{i=1}^{n}(1 - \frac{\mathrm{ED}(\mathrm{Output}[i], \mathrm{Answer}[i])}{\max(|\mathrm{Output}[i]|, |\mathrm{Answer}[i]|)})}{n}$, где как ED*(A, B)* обозначено редакционное расстояние между строками *A* и *B*. Отметим, что значения этой функции лежат в пределах от 0 до 1, при этом, чем «лучше» автомат соответствует тестам, тем больше значение функции приспособленности.

Для того чтобы выделять особи, проходящие все тесты, и те, которые проходят только часть, предлагалось вычислять функцию приспособленности для тестов по формуле: $\mathrm{FF_{test}} = \begin{cases} 0.5 \cdot \mathrm{T} \cdot \mathrm{FF_1}, \mathrm{FF_1} < 1 \\ \mathrm{T}, \mathrm{FF_1} = 1 \end{cases}$, где T – «стоимость» прохождения всех тестов.

Вклад *LTL*-формул в общую функцию приспособленности предлагается оценивать как доля верных формул рассматриваемой особи. Этот вклад можно оценить как $\mathrm{FF_{LTL}} = \mathrm{F} \cdot \frac{\mathrm{n_1}}{\mathrm{n_2}}$, где F – «стоимость» выполнения всех формул, $\mathrm{n_1}$ – число успешно выполненных *LTL*-формул, а $\mathrm{n_2}$ – общее число формул. Таким образом, чем больше число верных формул, тем больше вклад теморальных свойств в функцию приспособленности, а, при выполнимости всех свойств, их вклад становится равным F.

Функция приспособленности зависит не только от того, насколько «хорошо» автомат работает на тестах и удовлетворяет формулам, но и числа переходов, которые он содержит. Таким образов, функцию приспособленности можно вычислить по формуле: $\mathrm{FF} = \mathrm{FF_{test}} + \mathrm{FF_{LTL}} + \frac{(C - \mathrm{cnt})}{C}$, где как cnt обозначено число переходов в автомате, C – число, больше чем число переходов, а $\mathrm{FF_{test}}$ и $\mathrm{FF_{LTL}}$ – вклады тестов и формул соответственно. Эта функция приспособленности устроена таким образом, что при одинаковом значении $\mathrm{FF_{test}} + \mathrm{FF_{LTL}}$, отражающем «прохождение» тестов и *LTL*-формул автоматом, преимущество имеет модель, содержащая меньше переходов.

В тоже время, мы хотим построить модель, проходящую все тесты и удовлетворяющую всем *LTL*-формулам. Поэтому генетический алгоритм сначала строит такую особь, а потом уже пытается уменьшить число переходов. Для обеспечения такого поведения требуется подбирать значение *C* из формулы, приведенной выше, таким образом, что бы вклад успешно пройденного теста и удовлетворение темпоральному свойству был больше, чем, например, уменьшение числа переходов на единицу.

Также в настоящей работе предлагается не вычислять функцию приспособленности для всех тестов и формул одновременно, а разбивать их на группы и вычислять для каждой группы отдельно. Каждая такая группа включает набор тестов и набор *LTL*-формул, которые описывают определенное поведение модели, а значит особь, которая полностью проходит все тесты конкретной функциональности и удовлетворяет всем ее темпоральным свойствам, будет лучше, чем особь, проходящая только часть тестов из разных групп. Тогда можно записать функцию приспособленности следующим образом: $\mathrm{FF} = \sum_{i=1}^{N}\left(\mathrm{FF_{test,i}} + \mathrm{FF_{LTL,i}}\right) + \frac{(\mathrm{C} - \mathrm{cnt})}{\mathrm{C}}$, где $\mathrm{FF_{test,i}}$ и $\mathrm{FF_{LTL,i}}$ – функции приспособленности для

тестов и для темпоральных свойств, вычисленные для i-ой группы, а как N обозначено число групп. Вычисление функций приспособленности для каждой группы выполняется так же, как описано выше.

Оценим время вычисления функции приспособленности. Время вычисления редакционного расстояния пропорционально произведению длин последовательностей, для которых оно вычисляется. Таким образом, время вычисления функции приспособленности для тестов есть $O(\sum_{i=1}^{n} |\mathrm{Output}[i]| \cdot |\mathrm{Answer}[i]|)$. Заметим также, что добавление в набор тестов «префиксов» тестов не увеличивает время вычисления функции приспособленности, так как достаточно вычислить редакционное расстояние только для «самых больших» тестов, а для их префиксов редакционное расстояние взять из вычисленной таблицы динамического программирования.

Оценить время верификации одной *LTL*-формулы можно только примерно. Если n – длина формулы, то в худшем случае будет построен автомат Бюхи с $n \times 2^n$ состояниями [13]. Значит, автомат-пересечение будет содержать $n \times 2^n \times V$ состояний, где V – число вершин в модели. Заметим, что верификатор использует средство *LTL2BA* [14] для построения автомата Бюхи по *LTL*-формуле, которое преобразует формулу перед построением автомата и модифицирует автомат после. В итоге построенный автомат не будет экспоненциально расти от длины формулы. Как правило, верифицируемые формулы достаточно компактные, что не будет приводить к автоматам с большим числом состояний.

Так как формулы задаются заранее, то их преобразование в автоматы Бюхи производится один раз. Однако пересечения этих автоматов с автоматом модели придется выполнять каждый раз при вычислении функции приспособленности новой полученной особи.

Также заметим, что при верификации не всегда требуется целиком обходить автомат пересечения. Если модель не удовлетворяет темпоральному свойству, то контрпример может быть найден задолго до обхода всего автомата. Но даже, когда формула выполняется, автомат пересечения может быть меньше, чем $n \times 2^n \times V$, так как многие вершины могут оказаться недостижимы.

Из сказанного следует, что процесс верификации может занимать много времени и, чем больше формул мы хотим использовать при построении автомата, тем дольше будет вычисляться функция приспособленности конкретной особи. Так как в каждом поколении могут быть тысячи особей, то выбор верификатора и его эффективность в плане производительности крайне важны.

2.2.1. Учет результата верификации при вычислении функции приспособленности

Вклад каждого теста оценивается по редакционному расстоянию между эталонной выходной последовательностью и выходной последовательностью, сгенерированной особью. Тем самым, каждый тест вносит не просто «0» или «1», показывая, что тест пройден на конечном автомате или нет, а некоторое вещественное число из отрезка [0; 1] (оба конца включительно).

Предлагается оценивать вклад каждой *LTL*-формулы аналогичным образом – сделать вклад каждой формулы не дискретным, а вещественным (из отрезка [0; 1]). Однако, используемый верификатор умеет только сообщать, что формула верна или приводить контрпример. В настоящей работе верификатор был расширен таким образом, чтобы можно было помечать переходы в процессе двойного обхода в глубину. В результате, когда во время первого обхода в глубину состояние конечного автомата покидается, все переходы, ведущие из него, помечаются как *проверенные*. Это означает, что они точно не лежат на пути, опровергающем *LTL*-формулу.

Предлагается в качестве вклада *LTL*-формул в функцию приспособленности брать отношение числа проверенных переходов к числу достижимых переходов. Формулу можно

записать как $FF_{LTL} = \sum_{i=1}^{n} \frac{t_i}{T} \Big/ n$, где n – число *LTL*-формул, T – число достижимых переходов в особи, t_i – число переходов, помеченных как проверенные при верифкации i-ой формулы. Чем больше переходов было отмечено в процессе верификации, тем больше вклад формулы в функцию приспособленности, а значит и особь является более приспособленной.

Приведем пример вычисления вклада одной из *LTL*-формул. Пусть мы верифицируем конечный автомат из шести состояний, представленный на рис. 5. Цифрой «1» выделена часть автомата, на которой не удалось обнаружить контрпример, а цифрой «2» – часть автомата, на которой формула опровергается.

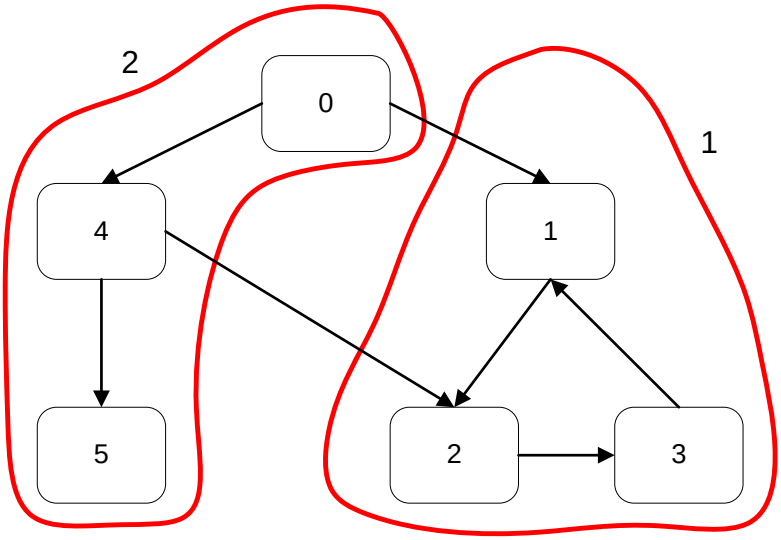


Рис. 5. Пример вычисления вклада LTL-формулы в функцию приспособленности

Таким образом, вклад данной формулы будет $FF_i = \frac{3}{7}$, где 3 – число помеченных переходов из первого подмножества, а 7 – общее число переходов в конечном автомате.

Заметим, что порядок обхода в глубину состояний и переходов автомата не гарантируется, поэтому могло получиться так, что контрпример (помеченный цифрой 2 на рис. 5) мог быть найден сразу. Тогда вклад формулы в функцию приспособленности оказался бы нулевым, так как ни один из переходов не был бы помечен.

2.3. Операция мутации

Выполнимость или невыполнимость *LTL*-формул позволяет только отбирать «лучшие» особи, но не позволяет «улучшать» популяцию путем скрещивания или мутации, так как этот процесс был бы случайным и не гарантировал бы увеличения числа верных утверждений в следующем поколении. То есть, учитывая только относительное число выполненных темпоральных свойств, мы не можем влиять на результат скрещивания или мутации, однако именно эти операции и приводят к росту значения функции приспособленности.

Для преодоления описанного недостатка при выполнении операции мутации предлагается учитывать контрпример, построенный верификатором. Такой контрпример представляет собой список вершин и переходов автомата, которые опровергают *LTL*-формулу. По сути, это путь в автоматной модели, на котором выполняется отрицание формулы. Имея информацию о таком пути, мы можем увеличить вероятность скрещивания или мутации для одного или нескольких переходов из контрпримера. Например, при выполнении мутации мы можем заменить конечное состояние перехода (входящего в контрпример), входное событие и выходные воздействия. Благодаря предложенным

действиям перестанет существовать данный контрпример для *LTL*-формулы, и увеличатся шансы новой особи соответствовать большему числу *LTL*-формул.

Если писать конкретнее, то при верификации выбирается самый длинный по числу переходов контрпример. Затем при создании новой особи с переходами из этого контрпримера могут происходить следующие действия:

- при копировании перехода в новую особь с некоторой вероятностью у него одновременно меняются: входное событие, число выходных воздействий и конечное состояние;
- при обычной мутации, если она заключается в удалении перехода у вершины, удаляется тот переход, который лежит на пути, опровергающем формулу.

Конечно, такие действия не гарантируют увеличение значения функции приспособленности у особи в следующем поколении. Но они хотя бы позволяют убрать путь, на котором выполняется отрицание *LTL*-формулы, а тогда, вероятно, исходная формула будет выполняться на всех возможных путях.

Мы не можем автоматически проанализировать семантику формулы и понять, как надо исправить переход, чтобы формула стала выполняться. Поэтому одновременно меняется входное событие, число выходных воздействий и конечное состояние перехода, так как мы не знаем, какой из предикатов в *LTL*-формуле выполняется или не выполняется на данном пути. Предикат может «говорить» о событии, может «говорить» о вызванном действии, а, может быть, рассматриваемый переход ведет в состояние, для которого некоторая подформула *LTL*-формулы является неверной.

Приведем пример верификации модели и операции мутации. Пусть одна из особей в поколении оказалась такой, как представлена на рис. 6.

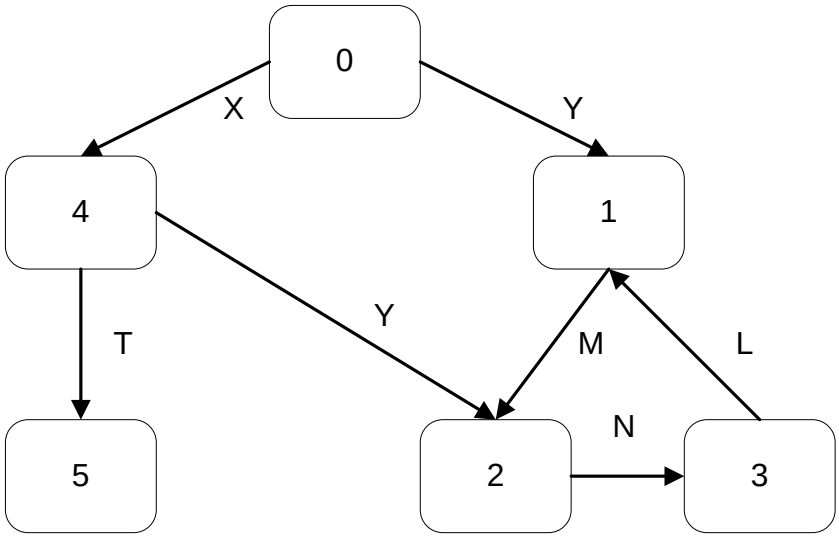


Рис. 6. Пример конечного автомата

Пусть необходимо проверить утверждение «G(!wasEvent(T))», которое означает, что никогда не будет обработано событие T. Так как алгоритм верификации заключается в поиске контрпримера, опровергающего формулу, то мы пытаемся найти путь, на котором выполняется отрицание формулы: «!G(!wasEvent(T))». По отрицанию формулы строится автомат Бюхи, представленный на рис. 7.

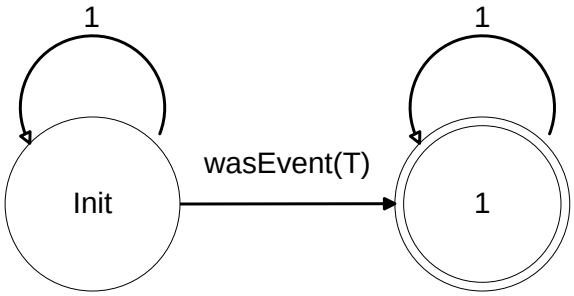


Рис. 7. Автомат Бюхи для формулы «!G(!wasEvent(T))»

Построенный автомат перейдет в допускающее состояние «1» в том и только в том случае, когда автомат модели совершит переход по событию T, так как только тогда предикат «wasEvent(T)» будет верен. Если же автомат модели не будет содержать такого перехода, или же он будет недостижим из начального состояния, то автомат Бюхи никогда не сможет перейти в состояние «1» и темпоральное свойство будет выполняться.

Не будем явно строить пересечение двух автоматов. Автомат Бюхи, построенный по отрицанию формулы, будет совершать переходы по петле из состояния «Init», пока будет обходиться автомат модели (состояния особи с номерами 0, 1, 2, 3, 4). Только после того, как автомат модели перейдет по событию T из состояния «4» в состояние «5», автомат, допускающий отрицание *LTL*-формулы, сможет сделать переход из состояния «Init» в состояние «1». Так как из состояния модели с номером 5 переходов больше нет, то автомат пересечения запустит второй обход в глубину. В результате работы верификатора будет найден путь в модели, опровергающий формулу (рис. 8).

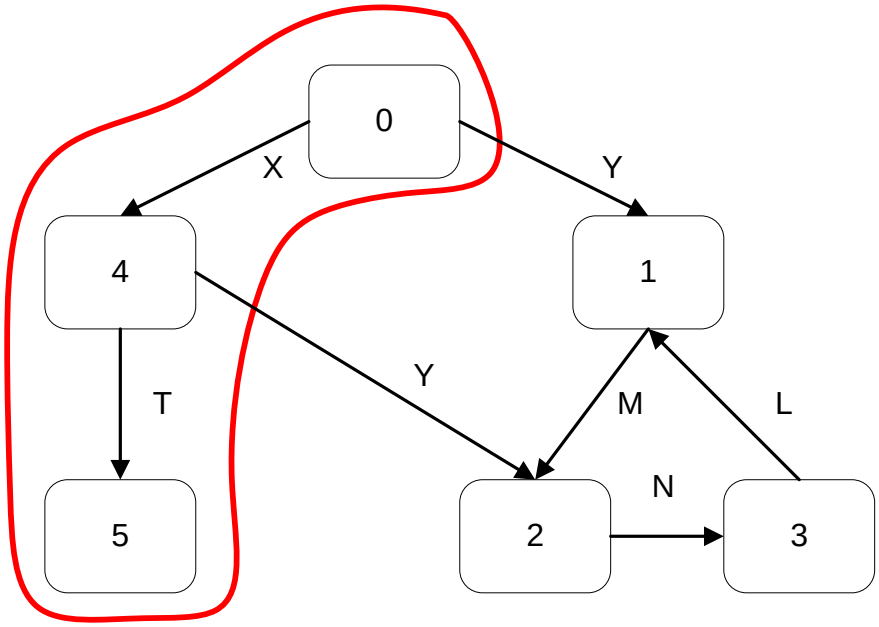


Рис. 8. Путь в модели, опровергающий LTL-формулу

Таким образом, если мы удалим любой переход в найденном контрпимере, переход по событию T перестанет существовать или быть достижимым. В то же время, может помочь и мутация, изменяющая событие на переходе, но, в нашем случае, мутировать должен именно переход по событию T.

Конечно, модель может оказаться не такой простой, и мутация окажется не такой эффективной. Например, если бы был переход из состояния модели «2» в состояние «5» по событию T, то сначала мог быть найден путь, проходящий через состояния с номерами 0, 1, 2, 5 (рис. 9). В таком случае мутация устранила бы данный контрпример, а формула все равно не стала бы выполняться. Но в следующем поколении, верификатор найдет путь,

проходящий по состояниям с номерами 0, 4, 5 и устранит второй контрпример, тем самым за два поколения найдется особь, удовлетворяющая заявленной формуле.

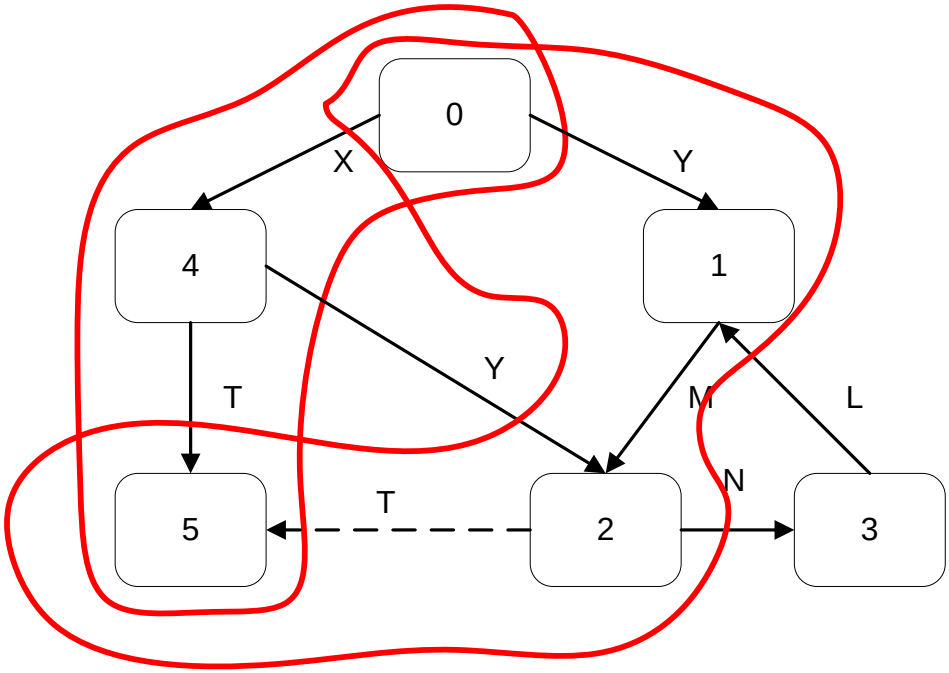


Рис. 9. Модель с двумя контрпримерами

В общем случае мутация остается случайным процессом, просто вероятность мутации переходов из контрпримера выше, чем остальных. Не стоит забывать, что, устраняя контрпример, мы можем нарушить прохождение какого-нибудь теста.

2.4. Операция скрещивания

Операция скрещивания может выполняться одним из трех способов: случайное, по тестам и с учетом результата верификации. Первые два способа описаны в работе [5].

Случайное скрещивание самое простое – выбираются две особи и их списки переходов «смешиваются». В результате получается особь, в которой часть переходов от одного родителя, а часть от другого.

Скрещивание по тестам основано на том, что часть конечного автомата, на которой «хорошо» проходятся тесты, переходит в новую особь без изменений, а остальные переходы «смешиваются», так же как при случайном. Такое скрещивание позволяет сохранить часть автомата, на которой проходятся тесты, тем самым не уменьшив функцию приспособленности.

2.4.1. Скрещивание с учетом результата верификации

Как было изложено выше, алгоритм верификации конечных автоматов основан на двойном обходе в глубину. Таким образом, когда первый обход в глубину покидает состояние, то подграф, образованный просмотренными состояниями и переходами, соответствует *LTL*-формуле.

Напомним, что если состояние в автомате-пересечении является допускающим, то второй обход в глубину проверяет, лежит ли данное допускающее состояние в сильной компоненте связанности. Если состояние не допускающее или цикл, проходящий через него, не был обнаружен, то состояние покидается. Так как алгоритм верификации использует обход в глубину, то все достижимые состояния из покидаемого так же просмотрены и мы можем пометить все исходящие переходы как проверенные.

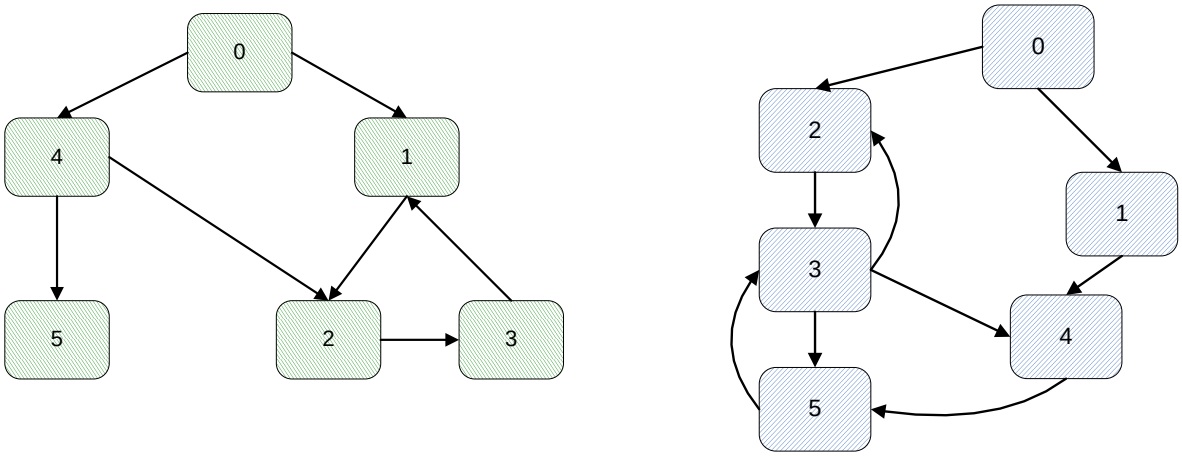
Приведем псевдокод предлагаемого метода, жирным шрифтом выделен фрагмент, отличающий его от обычного двойного обхода в глубину, помечающий исходящие переходы как верифицированные:

```
procedure emptiness
    for all q0 ϵ Q0 do
        dfs1(q0);
    terminate(False);
end procedure

procedure dfs1(q)
    local q’;
    hash(q);
    for all последователей q’ вершины q do
        if q’ не содержится в хэш-таблице then dfs1(q’);
    if accept(q) then dfs2(q);
    for all переходы t из вершины q do
        markAsVerified(t);
end procedure

procedure dfs2(q)
    local q’;
    flag(q);
    for all последователей q’ вершины q do
        if q’ в стеке dfs1 then terminate(True);
        else if q’ не является помеченной then dfs2(q’);
        end if;
end procedure
```

После пометки переходов, как соответствующих темпоральному свойству, можно проводить скрещивание таким образом, чтобы помеченные переходы перешли в новую особь без изменений.

Так как обычно проверяются несколько формул, то при скрещивании можно брать либо объединение помеченных переходов для разных формул, либо пересечение. Возможно, что какая-то формула выполняется и все переходы помечены, тем самым объединением помеченных переходов будут все переходы. В то же время, мы не знаем в какой последовательности проверяются переходы, и контрпример может быть найден сразу для неверной формулы. Тем самым, ни один переход не будет помечен, и пересечение будет пусто. Что бы избежать обе эти проблемы предлагается брать объединение (или же пересечение) не для всех формул, а только для случайной их выборки.

Приведем пример такого скрещивания. Предположим, что каждая особь популяции содержит по шесть состояний, они отмечены номерами от 0 до 5 (рис. 10).



а б

Рис. 10. Две особи, построенные в ходе генетического алгоритма

Каждая из таких особей не удовлетворяет всем *LTL*-формулам, но, помечая переходы в процессе верификации, можно получить некий подграф из переходов и состояний, на котором часть формул выполняется. Данный подграф перейдет в новую особь без изменений, а остальные переходы случайным образом перемешаются.

Такое скрещивание приведено на рис. 11, выделенная часть подграфа, образована помеченными переходами.

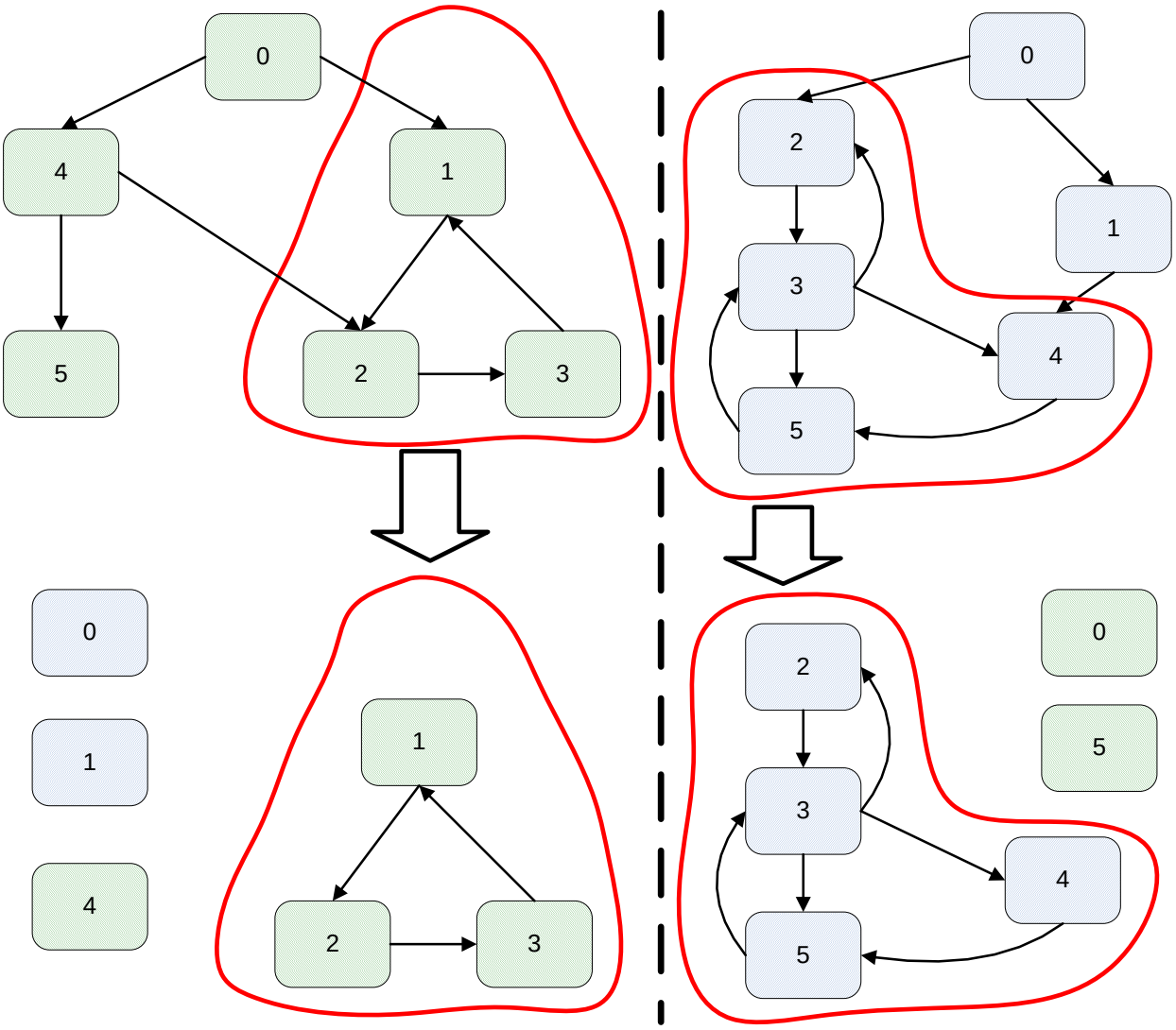


Рис. 11. Скрещивание с учетом результата верификации

В результате такого скрещивания, часть конечного автомата, на которой часть формул выполняется, сохранится, что позволит увеличить функцию приспособленности.

2.5. Методика построения автоматных программ

Для сравнения методов построения модели на основе тестов и на основе одновременно тестов и темпоральных свойств, сначала приведем методику построения автоматных программ только на основе тестов (рис. 12).

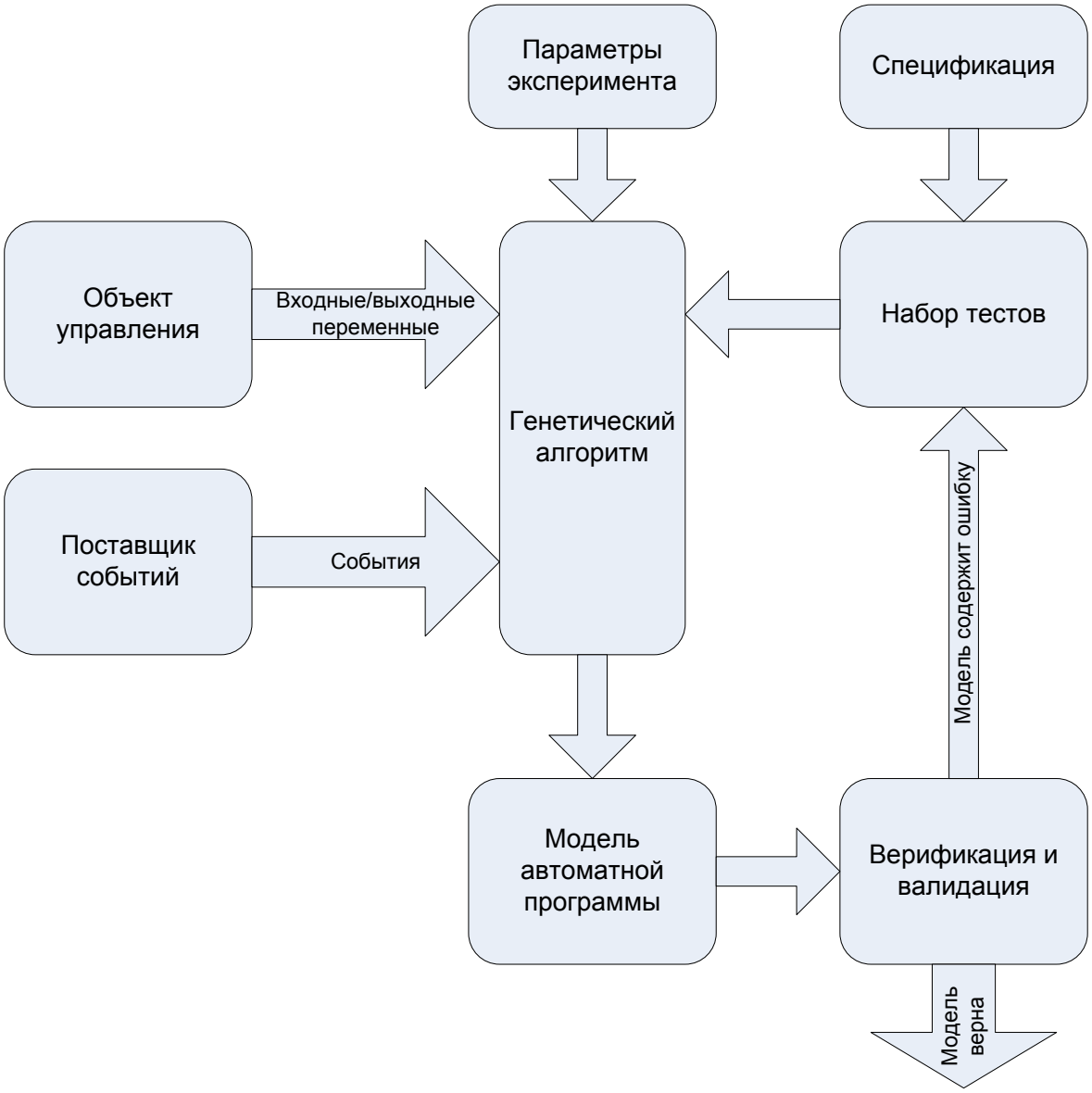


Рис. 12. Методика построения автоматных программ на основе тестов с помощью генетических алгоритмов

На вход генетическому алгоритму подаются списки входных переменных, выходных переменных и событий. Входные и выходные переменные берутся из «объекта управления», который либо реализован заранее, либо известны его методы. События, которые обрабатываются моделью, берутся из «поставщика событий», который также либо реализован заранее, либо известны только возможные события. В общем случае можно использовать несколько объектов управления и несколько поставщиков событий.

Далее по спецификации программы или из других априорных знаний о ее поведении на основе известных переменных и событий строится набор тестов. При записи тестов используются события и входные переменные в качестве входных данных теста (Input[i]), а выходные переменные, как эталонная последовательность выходных данных (Answer[i]).

На вход генетическому алгоритму также подаются параметры эксперимента, такие как размер популяции, вероятность мутации, число поколений до «большой» и «малой» мутации и т.д.

Далее генетический алгоритм строит автоматную модель. Отметим, что в этом методе генетический алгоритм одинаков для всех задач – для каждой новой задачи необходим только новый набор тестов, входных и выходных воздействий и, возможно,

параметров алгоритма. Построенная модель проходит все тесты, однако мы не можем быть уверены в ее правильности, поэтому необходима дополнительная верификация и валидация.

Если модель оказалась верной, то ее можно использовать в реальной системе. Однако, если верификатор обнаружил несоответствие модели и спецификации, то необходимо добавлять новые тесты во входной набор и выполнять построение заново. В худшем случае, мы никогда не сможем построить корректную модель, так как нет гарантии, что такой цикл разработки модели когда-нибудь завершится.

Заметим, что мы можем вручную модифицировать модель, чтобы она стала соответствовать спецификации, однако это может оказаться сложнее, чем просто создание модели вручную.

В настоящей работе предлагается использовать верификацию на стадии создания программы. Методика работы предлагаемого метода представлена на рис. 13.

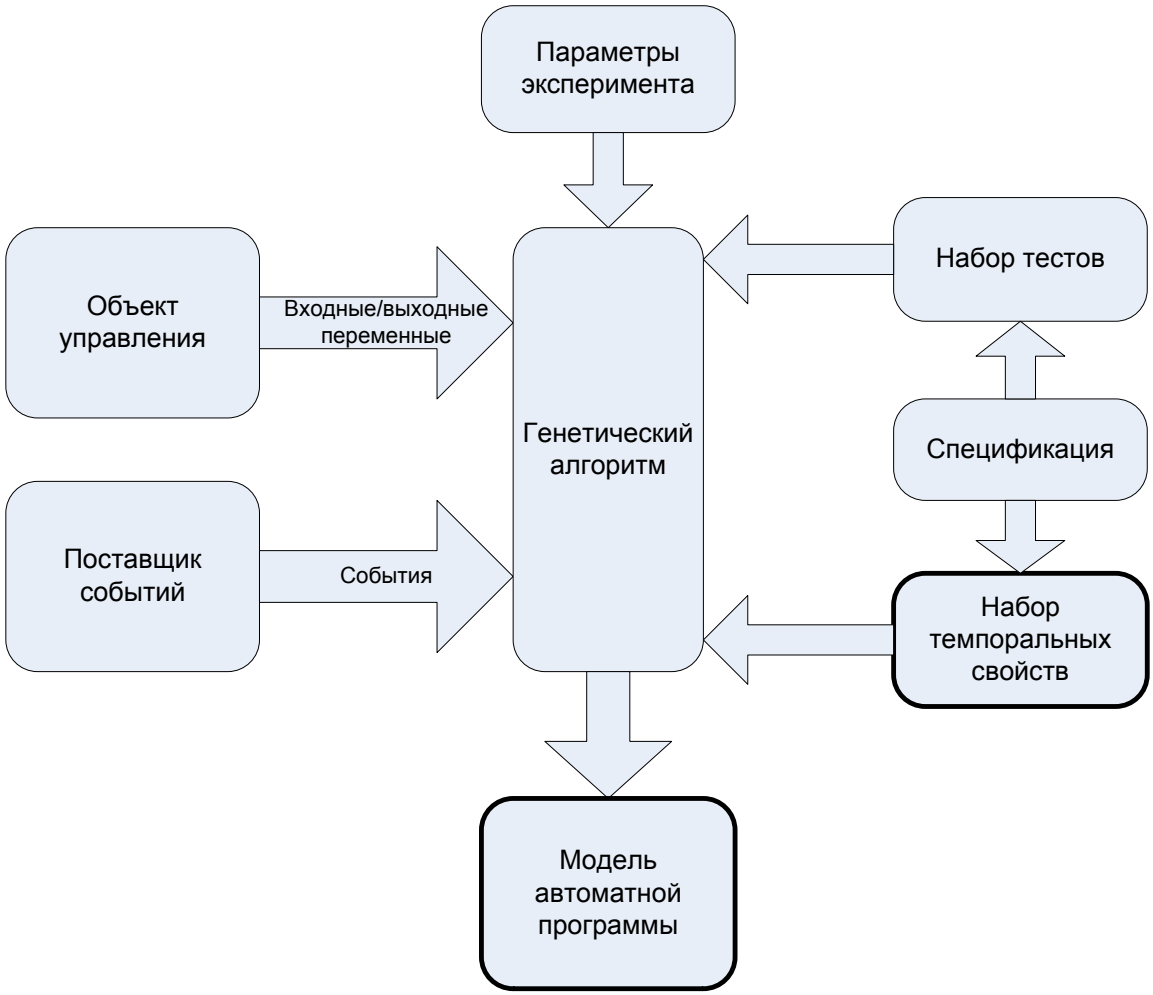


Рис. 13. Методика создания автоматных программ на основе тестов и темпоральных свойств

В отличие от метода, основанного только на тестах, входными данными являются также темпоральные свойства. Темпоральные свойства формулируются на основе спецификации и записываются в виде *LTL*-формул. Как видно из схемы, представленной на рис. 13, построенная модель автоматной программы не требует дополнительной верификации, так как каждая особь в каждом поколении верифицировалась, и мы выбрали в качестве результата работы метода ту, у которой была максимальная функция приспособленности. Это означает, что она проходит все тесты и удовлетворяет всем темпоральным свойствам, а значит она соответствует спецификации и дополнительные проверки не требуются.

Таким образом, если набор тестов и набор теморальных свойств непротиворечивы, то автоматная модель будет построена, и она будет соответствовать заявленной спецификации.

Выводы по главе 2

В разделе описан алгоритм построения конечного автомата на основе тестов и *LTL*-формул. Дано описание представления конечного автомата в виде хромосомы особи, описаны операции скрещивания и мутации. Также предлагается учитывать результат верификации при вычислении функции приспособленности.

В последней части настоящего раздела описана методика построения конечных автоматов только на основе тестов и на основе тестов и темпоральных свойств одновременно. Показаны различия в двух методиках.

ГЛАВА 3. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ МЕТОДА И ЭКСПЕРИМЕНТАЛЬНОЕ ИССЛЕДОВАНИЕ

В настоящей главе описана программная реализация предлагаемого метода, и приводятся примеры построения управляющих конечных автоматов. Данный метод тестировался на двух примерах. Первый заключался в построении конечного автомата, управляющего часами с будильником. Во время второго эксперимента выполнялось построение конечного автомата, управляющего дверьми лифта.

3.1. Программная реализация

Предлагаемый метод реализован на языке *Java* и является дополнением к реализации из работы [5]. К предыдущей работе были добавлены два пакета: для чтения тестов, формул и для их верификации. Диаграмма пакетов приведена на рис. 14.

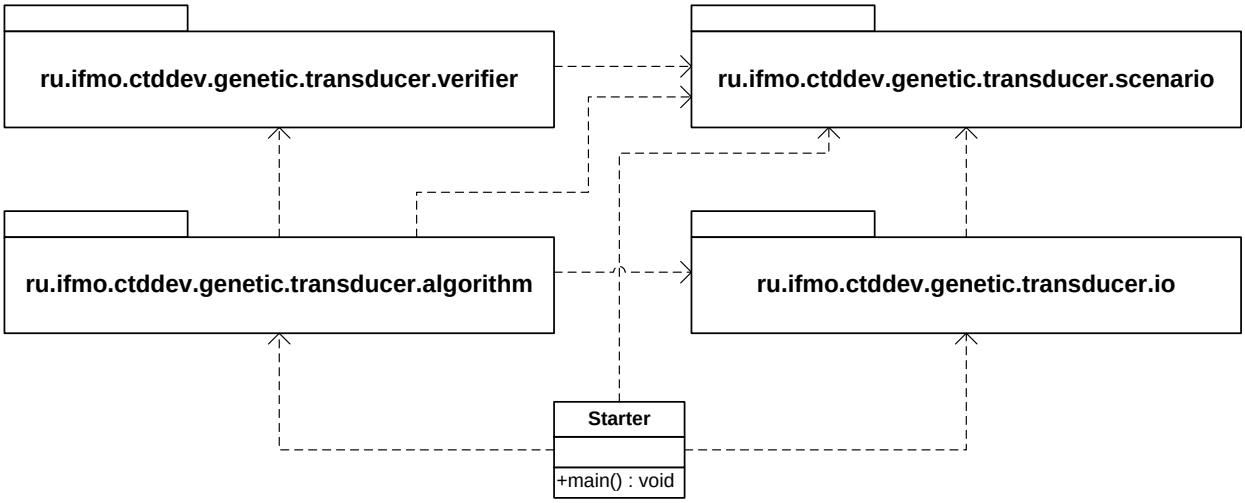


Рис. 14. Диаграмма пакетов программы

В пакет `ru.ifmo.ctddev.genetic.transducer.scenario` добавлен класс `TestGroup`, который хранит набор тестов и *LTL*-формул для конкретной группы. Теперь генетический алгоритм на вход получает не список тестов, а список объектов данного класса.

Класс `FitnessCalculator` реализует вычисление функции приспособленности на основе редакционного расстояния и относительного числа успешно выполненных формул.

```
public double calcFitness(FST fst) {
  fst.doLabelling(this.tests);

  verifier.configureStateMachine(fst.getStates(),
                                 fst.getInitialState());
  int[] verRes = verifier.verify();

  double res = 0;
  int i = 0;
  for (TestGroup group : groups) {
    double sum = 0;
    int cntOk = 0;

    for (Path p : group.getTests()) {
      String[] output = fst.transform(p.getInput());
      String[] answer = p.getOutput();
      double t;
```

```
        if (output == null) {
          t = 1;
        } else {
          t = editDistance(output, answer)
              / Math.max(output.length, answer.length);
        }
        sum += 1 - t;
        if (same(output, answer)) {
          cntOk++;
        }
      }
      int testsSize = group.getTests().size();
      int formulasSize = group.getFormulas().size();

      if (testsSize > 0) {
        res += (cntOk == testsSize)
            ? TESTS_COST
            : 0.5 * TESTS_COST * (sum / testsSize);
      }
      if (formulasSize > 0) {
        res += FORMULAS_COST * verRes[i] / formulasSize;
      }
      i++;
    }
    return res + 0.01 * (100 - fst.getTransitionsCount());
  }
```

Сначала выполняется алгоритм расстановки пометок [5], после этого полученный автомат с проставленными выходными воздействиями верифицируется, и определяется число успешно «пройденных» формул для каждой группы. Также при верификации помечаются переходы, принадлежащие самому длинному контрпримеру. После этого вычисляется редакционное расстояние для тестов в каждой группе, и суммируются вклады тестов и темпоральных свойств.

Класс `FST`, реализующий конечный автомат Мили, хранит номер начального состояния, число состояний, набор переходов для каждого состояния, набор входных событий и набор выходных воздействий.

```
private final int initialState;
private final int stateNumber;
private Transition[][] states;

private final String[] setOfInputs;
private final String[] setOfOutputs;
```

В этом классе также реализованы операции мутации, скрещивания и алгоритм расстановки пометок. Их реализацию можно уточнить в работе [5]. В настоящей же работе в данный класс было добавлено удаление перехода из пути при мутации, опровергающего *LTL*-формулу.

Класс `Transition` представляет собой переход в автомате, он хранит входное воздействие, число выходных воздействий и номер состояния, в которое перейдет автомат по этому переходу. Класс представляет набор методов, но в данной работе интересен тот, который копирует переход в новую особь при скрещивании. Он реализован, как описано в разд. 2.3:

```
public Transition copy(String[] setOfInputs,
```

```
                        String[] setOfOutputs,
                        int stateNumber) {
    if (isUsedByVerifier()
            && !used
            && (RANDOM.nextDouble() < 0.1)) {
        return new Transition(
            setOfInputs[RANDOM.nextInt(setOfInputs.length)],
            mutateOutputSize(outputSize, setOfOutputs.length),
            RANDOM.nextInt(stateNumber));
    }
    return new Transition(input, outputSize, newState);
}
```

В добавленном пакете `ru.ifmo.ctddev.genetic.transducer.io` расположены классы для чтения тестов и *LTL*-формул из *xml*-файла и записи модели в файл в формате *UniMod* [16].

Класс `TestsReader` позволяет читать из `xml`-файла параметры алгоритма (вероятность мутации, размер популяции, ожидаемое число состояний, число поколений до мутации и т.д.), набор тестов и темпоральных свойств, разбитых по группам. Класс `OneGroupTestsReader` позволяет читать входные данные для метода, игнорируя разбивку по группам (создавая всего одну группу).

Класс `UnimodModelWriter` сохраняет особь в файл в формате *UniMod* `xml`-описания модели. В результате, созданную автоматную модель можно сразу запускать в инструментальном средстве *UniMod*, при условии, что реализованы поставщики событий и объекты управления.

В пакете `ru.ifmo.ctddev.genetic.transducer.verifier` находятся классы, относящиеся к верификации, трансляции формул в автомат Бюхи и преобразованию особи генетического алгоритма в объект-автомат, принимаемый верификатором.

Класс `ModifiableAutomataContext` является представлением верифицируемой автоматной программы. Он содержит объект управления, поставщик событий и конечный автомат модели. Его особенность в том, что конечный автомат может меняться, так как у каждой особи он свой, в то время как поставщик событий и объект управления остаются неизменными и создаются один раз за время генерации конечного автомата. Приведем поля из данного класса, они являются интерфейсами, описанными в используемом верификаторе.

```
private IControlledObject co;
private IEventProvider ep;
private IStateMachine<IState> machine;
```

Класс `VerifierFactory` предоставляет методы для трансляции формул в автомат Бюхи, для представления особи в виде автомата, принимаемого используемым верификатором, и метод для верификации. Причем трансляция формул происходит только один раз при запуске программы, так как формулы в процессе работы генетического алгоритма не меняются, а значит, их можно преобразовывать в автомат Бюхи только один раз. После верификации модели соответствующий метод возвращает число пройденных тестов для каждой группы и помечает самый длинный путь в автомате, являющийся контрпримером для одной из формул.

```
void prepareFormulas(TestGroup[] groups);
void configureStateMachine(FST fst);
int[] verify();
```

Вызов верификации предполагает, что модель была преобразована в модель, принимаемую на вход верификатором, для этого необходимо вызвать метод

`void configureStateMachine(FST fst).` После этого автомат верифицируется методом `int[] verify()`. Который возвращает число верных формул для каждой из группы. Приведем код данного метода:

```
public int[] verify() {
  int[] res = new int[preparedFormulas.length];
  IVerifier<IState> verifier = new SimpleVerifier<IState>(
      context.getStateMashine(null).getInitialState());
  List<IIntersectionTransition> longestList =
      Collections.emptyList();

  for (int i = 0; i < preparedFormulas.length; i++) {
    int formulasOk = 0;

    for (IBuchiAutomata buchi : preparedFormulas[i]) {
      List<IIntersectionTransition> list =
          verifier.verify(buchi, predicates);
      if ((list == null) || (list.size() == 0)) {
        formulasOk++;
      } else {
        if (longestList.size() < list.size()) {
          longestList = list;
        }
      }
    }
    res[i] = formulasOk;
  }

  for (IIntersectionTransition t : longestList) {
    AutomataTransition trans =
        (AutomataTransition) t.getTransition();
    if ((trans != null)
        && (trans.getAlgTransition() != null)) {
      trans.getAlgTransition().setUsedByVerifier(true);
    }

  return res;
}
```

Класс `AutomataTransition` является представлением перехода для верификатора. Также данный класс сохраняет ссылку на переход в особи, чтобы по контрпримеру верификатора совершить обратное преобразование в переход модели.

3.2. Построение конечного автомата управления часами с будильником

Так же, как и в работе [5], метод исследовался на задаче построения конечного автомата, управляющего часами с будильником [15]. Эти часы имеют три кнопки (рис. 15), которые служат для изменения режима их работы и для настройки текущего времени или времени срабатывания будильника.

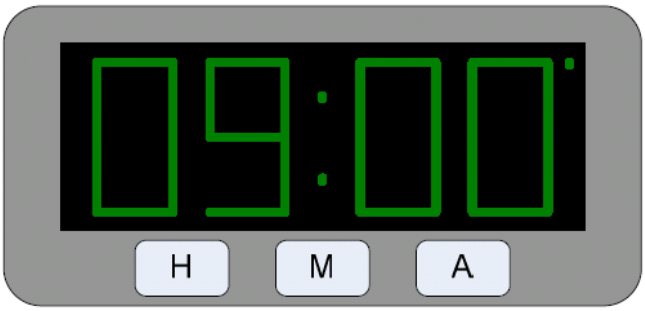


Рис. 15. Внешний вид часов с будильником

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

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

Напомним, что конечный автомат часов имеет четыре входных события:

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

Он также содержит две входные переменные:

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

Кроме этого автомат имеет семь выходных воздействий:

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

Поведение часов с будильником может быть описано графом переходов конечного автомата, который приведен в [15] и построен вручную (рис. 16).

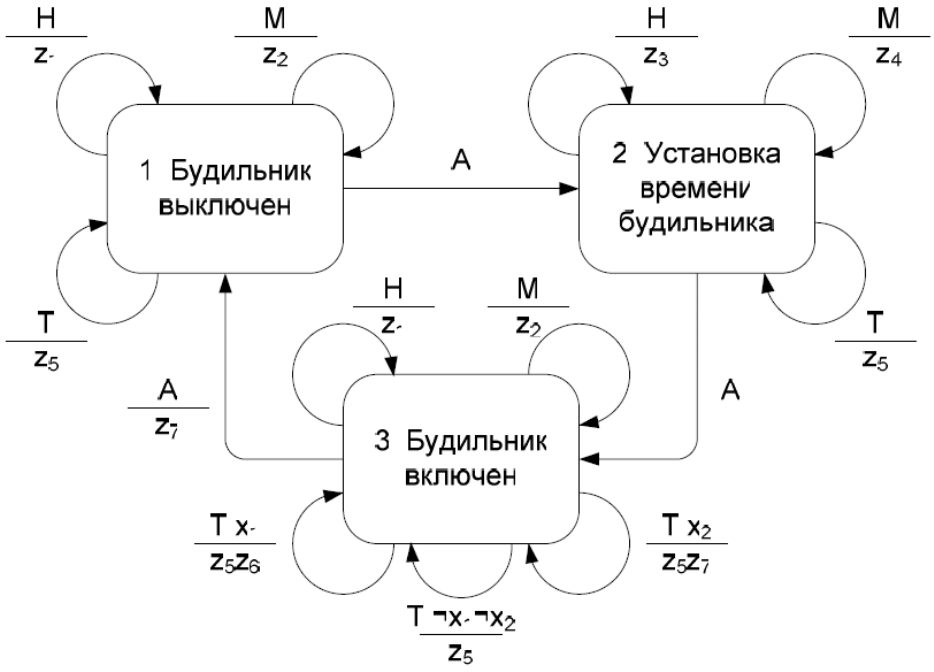


Рис. 16. Граф переходов конечного автомата управления часами с будильником

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

3.2.1. Система тестовых примеров и темпоральных свойств

В систему тестов для построения автомата управления часами с будильником входят 38 тестов, они подробно описаны в работе [5]. Данные тесты применялись совместно с темпоральными свойствами. Тесты преднамеренно не менялись и не добавлялись новые, чтобы можно было оценить, какой вклад дает верификация. Поэтому для настоящей работы представляют интерес *LTL*-формулы, которые проверялись верификатором при построении конечного автомата. Их описание приведено в табл. 1.

Таблица 1. Темпоральные свойства часов с будильником

Формула	Комментарий
G(wasEvent(T) <=> wasFirstAction(z5))	Обработка события T равносильна тому, что первым было вызвано действие z5. В терминах модели утверждение можно записать как: срабатывание таймера приводит к прибавлению минуты к текущему времени, и прибавление минуты может быть только по срабатыванию таймера.
G(wasEvent(H) <=> [wasAction(z1) or wasAction(z3)])	Обработка события H равносильна вызову действия z1 или z3. В терминах модели: нажатие кнопки «часы» приводит к увеличению текущего времени на один час или времени будильника на один час, и увеличение времени на один час может быть только по нажатию соответствующие кнопки.
!F(wasAction(z1) and wasAction(z3))	Неверно, что в будущем z1 и z3 могут быть одновременно. В терминах модели: нельзя одновременно увеличить время часов и время будильника на час.
G(wasEvent(M) <=> [wasAction(z2) or wasAction(co.z4)])	Обработка события M равносильна вызову действия z2 или z4. В терминах модели: нажатие кнопки «минуты» приводит к увеличению на одну минуту времени часов или времени будильника, и увеличение времени на минуту может быть только по нажатию соответствующей кнопки.
!F(wasAction(z2) and wasAction(z4))	Неверно, что в будущем z2 и z4 могут быть одновременно. В терминах модели: нельзя одновременно увеличить время часов и время будильника на минуту.
G([wasEvent(A) and wasAction(z7)] => X(R[wasEvent(A), !wasAction(z3) and !wasAction(z4)]))	Если было обработано событие A и вызвано действие z7, то не могут быть вызваны действия z3 и z4, пока не произойдет событие A. В терминах модели: если будильник был отключен, то нельзя выставлять время будильника, не нажав кнопу A.
G([wasAction(z3) or wasAction(z4)] => X(R[wasEvent(A), !wasAction(z6) and !wasAction(z7)]))	Если было вызвано действие z3 или z4, то не будут вызваны действия ни z6, ни z7 до тех пор, пока не будет обработано событие A. В терминах модели: если мы выставляем время будильника, то звонок не будет включен и не будет выключен, пока не будет нажата кнопка A.
G([wasAction(z1) or wasAction(z2)] => X(R[wasEvent(A), !wasAction(z3) and !wasAction(z4)]))	Если было вызвано действие z1 или z2, то нельзя вызвать z3 и z4 до события A. В терминах модели: если мы выставляем время часов, то нельзя выставлять время будильника, не нажав кнопку A.

G([wasAction(z3) or wasAction(z4)] => X(R[wasEvent(A), !wasAction(z1) and !wasAction(z2)]))	Если было вызвано действие z3 или z4, то нельзя вызывать z1 и z2 до события A. В терминах модели: если мы выставляем время будильника, то, не нажав A, нельзя выставлять время часов.
!F(wasAction(z6) and wasAction(z7))	Не могут быть одновременно вызваны действия z6 и z7. В терминах модели: нельзя одновременно включить и отключить сигнал будильника.
F(!(wasEvent(A) and !wasAction(z7)))	В будущем не верно, что произойдет A и не будет вызвано z7. В терминах: кнопка A не может нажиматься бесконечно, не отключив сигнал будильника, или что нажатие кнопки A рано или поздно отключит будильник.

Заметим, что предлагаемый подход не позволяет строить конечный автомат только на основе *LTL*-формул. Во-первых, тесты используются для расстановки выходных воздействий на переходах. Во-вторых, верификация плохо «улучшает» популяцию, а скрещивание по тестам позволяет передавать в новую особь часть «хороших» переходов. Все это приводит к более быстрому росту функции приспособленности, чем просто случайные мутации.

Таким образом, применение только темпоральных свойств приводит к слишком медленному росту функции приспособленности. Только совместное использование тестов и *LTL*-формул позволяет добиться прогнозируемого роста функции приспособленности и заранее заданного поведения построенной модели.

3.2.2. Результаты применения генетического алгоритма

Для сравнения различных способов построения конечных автоматов было проведено три варианта экспериментов. В первом случае автомат строился только на основе тестов, во втором на основе тестов и темпоральных свойств, когда верификация учитывалась только при вычислении функции приспособленности и мутации, в последнем эксперименте верификация использовалась на всех стадиях генетического алгоритма (вычисление функции приспособленности, скрещивании и мутация). Эксперимент каждого типа запускался по 1000 раз и для него строился один и тот же автомат управления часами с будильником, представленный в разделе 3.2. Входные параметры каждого эксперимента были идентичными.

Общие параметры экспериментов представлены в табл. 2.

Таблица 2. Параметры эксперимента построения автомата, управляющего часами с будильником

Параметр эксперимента	Значение
Размер начальной популяции	2000
Число состояний у автоматов в начальном поколении	4
Ожидаемое число переходов	14
Доля особей, переходящих в следующее поколение. Остальные будут получены с помощью кроссовера	10%
Число поколений до «малой» мутации	70
Число поколений до «большой» мутации	100
Вероятность мутации особи	1%

В результате всех трех экспериментов получался автомат изоморфный построенному вручную с точностью до названия состояний (рис. 17), в котором из начального (отмечено «жирной» рамкой) достижимы только три состояния из четырех. Если

удалить недостижимое состояние, то этот граф переходов будет изоморфен построенному вручную.

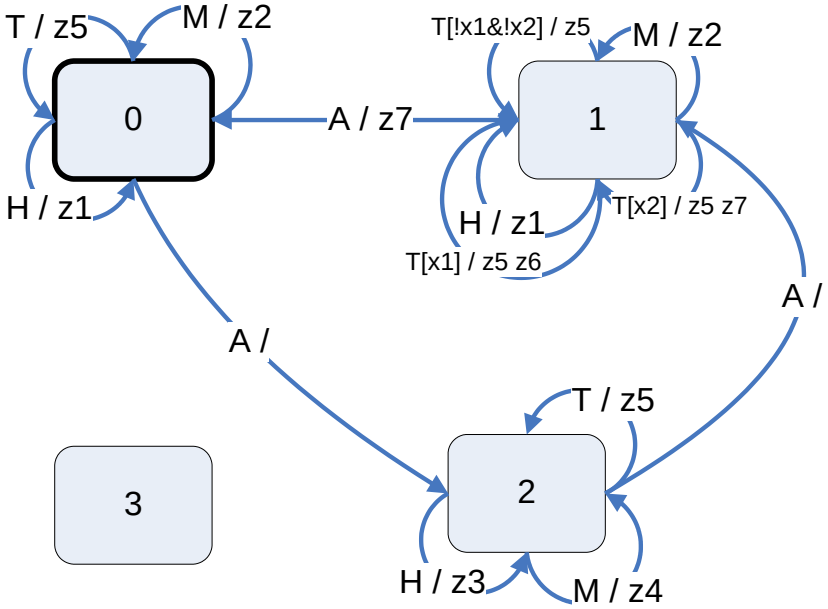


Рис. 17. Граф переходов автомата, построенного с помощью алгоритма генетического программирования

Эксперименты производились с разбивкой тестов и темпоральных свойств на группы. Всего было 4 группы тестов и 4 группы формул. Вклад тестов брался равным 100, а вклад *LTL*-формул – 10 (T = 100, F = 10, C=100 из разд. 2.2). Таким образом, вклад тестов в одной группе мог быть максимум равным 100, а формулы вкладывали максимум 10. Вклад числа переходов достигал максимума 0.86 на особях, проходящих все тесты и формулы.

Первый вид экспериментов заключался в построении автомата управления часами с будильником только на основе тестов. Число тестов было 38 штук, которые были взяты из работы [5]. Значение функции приспособленности построенного автомата было 400.86. В качестве численной оценки скорости работы алгоритма измерялось число вычислений функции приспособленности при нахождении автомата. Среднее значение вычислений функции приспособленности оказалось равным 1.45×10^6. Минимальное число вычислений – 2.561×10^5. Максимальное число – 9.24×10^6. Среднеквадратичное отклонение – 1.106×10^6. Плотность распределения вероятности числа вычислений функции приспособленности при построении автомата только на основе тестов представлена на рис. 18.

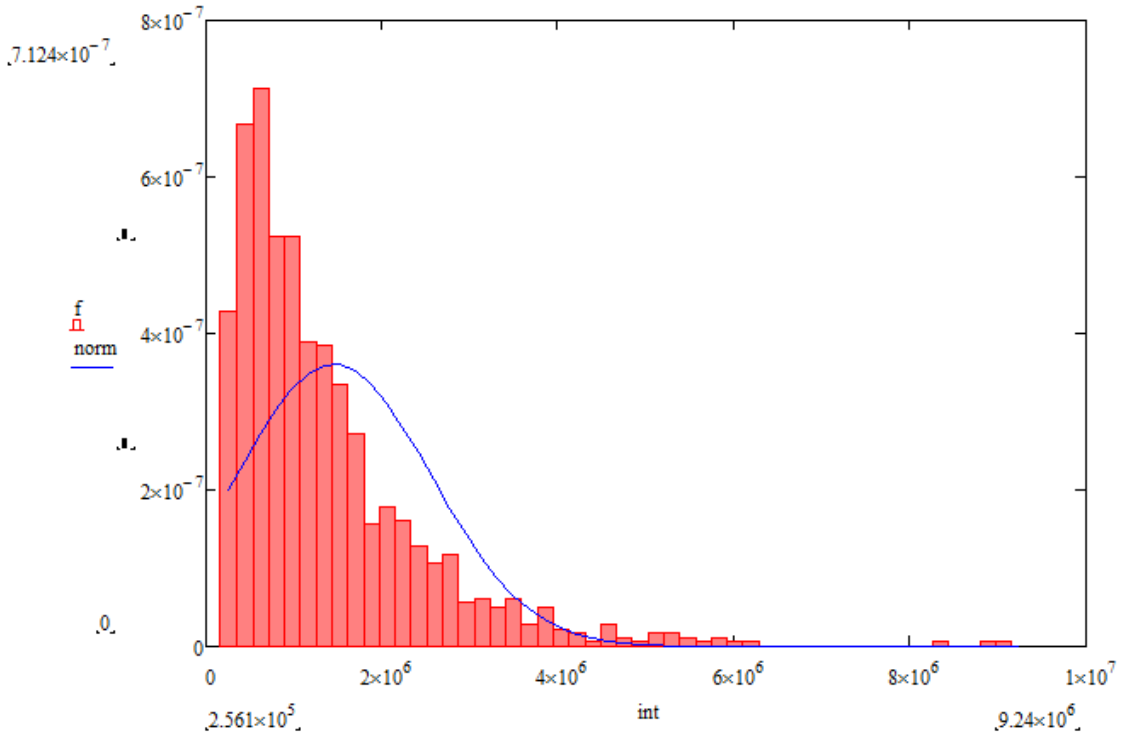


Рис. 18. Плотность распределения вероятности числа вычислений функции приспособленности при построении автомата только на основе тестов

Второй эксперимент заключался в построении автомата, управляющего часами с будильником на основе тестов и темпоральных свойств, когда верификация учитывалась при вычислении функции приспособленности и мутации. Вклад *LTL*-формул в функции приспособленности рассматривался как отношение выполнимых формул к общему числу формул. Темпоральные свойства были представлены *LTL*-формулами из раздела 3.2.1. Значение функции приспособленности построенного автомата – 440.86. Среднее значение вычислений функции приспособленности оказалось равным 2.425×10^6. Минимальное число вычислений – 2.182×10^5. Максимальное число – 1.949×10^7. Среднеквадратичное отклонение – 2.311×10^6. Плотность распределения вероятности числа вычислений функции приспособленности при построении автомата на основе тестов и *LTL*-формул представлена на рис. 19.

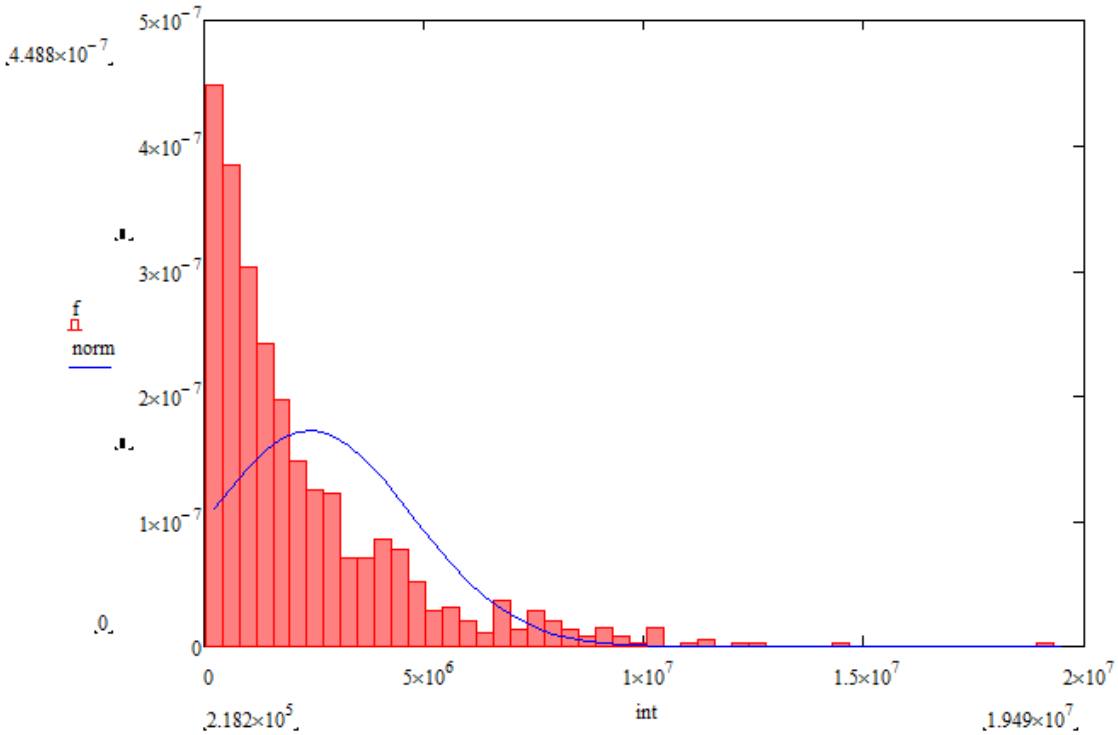


Рис. 19. Плотность распределения вероятности числа вычислений функции приспособленности на основе тестов и LTL-формул (верификация учитывалась на этапе вычисления функции приспособленности и мутации)

Третий эксперимент заключался в построении автомата, управляющего часами с будильником на основе тестов и темпоральных свойств, когда верификация учитывалась при вычислении функции приспособленности, скрещивании и мутации. Темпоральные свойства были представлены *LTL*-формулами из раздела 3.2.1. Значение функции приспособленности построенного автомата – 440.86. Среднее значение вычислений функции приспособленности оказалось равным 1.832×10^6. Минимальное число вычислений – 2.038×10^5. Максимальное число – 1.301×10^7. Среднеквадратичное отклонение – 1.662×10^6. Плотность распределения вероятности числа вычислений функции приспособленности в третьем эксперименте представлена на рис. 20.

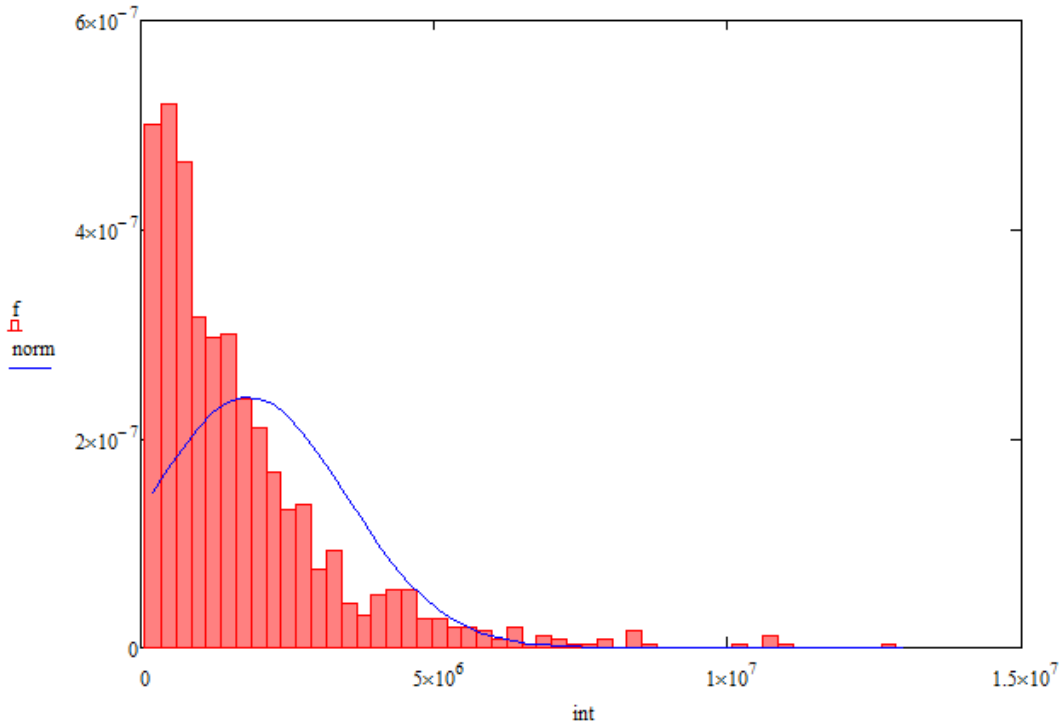


Рис. 20. Плотность распределения вероятности числа вычислений функции приспособленности на основе тестов и LTL-формул

Заметим, что число вычислений функции приспособленности для экспериментов с *LTL*-формулами оказалось выше. Хотя минимальное число вычислений, при котором был построен конечный автомат оказалось ниже, а среднеквадратичное отклонение, наоборот возросло. В то же время, учет верификации при скрещивании позволил ускорить процесс построения управляющего конечного автомата.

Замедление работы построения конечного автомата в плане числа вычислений функции приспособленности можно объяснить тем, что автомат на основе тестов и так получается «правильный». То есть, используя только тесты, метод строит модель, соответствующую спецификации, и она такая же, как и построенная вручную.

В верифицируемом наборе темпоральных свойств встречается много формул вида *G(A => B)*. Такие формулы выполняются всегда, если выражение *A* никогда не встречается. Значит новая особь, в которой стало выполняться выражение *A*, но не выполняется *B*, стала хуже родителя (для которого *A* не выполнялось никогда). Такая модель будет иметь меньшую функцию приспособленности, чем модель, из которой она образовалась. Но понятно, что такая модель может проходить все те же тесты, что и родитель и служить «прародителем» автомата, для которого формула снова станет выполняться.

Исходя из результатов эксперимента, можно сделать вывод, что в случае, когда тесты строят «правильный» автомат, использование верификации замедляет этот процесс. Но можно уменьшать влияние *LTL*-формул за счет параметра F (разд. 2.2) при вычислении функции приспособленности, так как, используя только тесты, мы все равно не можем быть уверенными, что автомат будет построен без ошибок. Это позволит нам оставаться уверенными в построенной модели, но не будет мешать построению автомата на основе тестов.

В случае же, когда только тесты не могут построить автомат, одновременно проходящий все тесты и удовлетворяющий всем темпоральным свойствам, или же модель строится неверной, можно увеличивать влияние формул в формуле вычисления функции приспособленности.

Заметим, что если бы модель на основе тестов строилась неправильной, то потребовалось бы удалить «надбавку» за прохождение всех тестов. То есть формулу для учета влияния тестов в таком случае можно записать $FF_{test} = T \cdot FF_1$. В противном случае мы получили бы «застревание» популяции в локальном максимуме – когда проходят все тесты, но формулы выполняются не все.

3.3. Построение конечного автомата управления дверьми лифта

Предложенный метод также исследовался на задаче построения конечного автомата управления дверьми лифта [8]. Двери лифта умеют открываться и закрываться. Если при закрытии дверей появилось препятствие, то требуется прекратить их закрывание и открыть двери. Как и любой настоящий лифт, рассматриваемый лифт может сломаться в процессе открывания или закрывания дверей, и тогда необходим звонок в аварийную службу.

Заметим, что после поломки лифта не предусмотрено возвращение автомата в работоспособное состояние, то есть в модели не предусмотрено такое событие, как «ремонт окончен» (в этом случае предполагается, что программа просто будет перезапущена).

Конечный автомат лифта имеет пять входных событий:

- *e11* – открыть двери (нажата кнопка «открыть двери»);
- *e12* – закрыть двери (нажата кнопка «закрыть двери»);
- *e2* – двери успешно открыты или закрыты;
- *e3* – препятствие мешает закрытию дверей;
- *e4* – двери лифта сломались.

Кроме этого автомат имеет три выходных воздействия:

- *z1* – начать открывание дверей;
- *z2* – начать закрывание дверей
- *z6* – звонок в аварийную службу.

Поведение дверей лифта может быть описано конечным автоматом, взятым из работы [8] и построенным вручную (рис. 21).

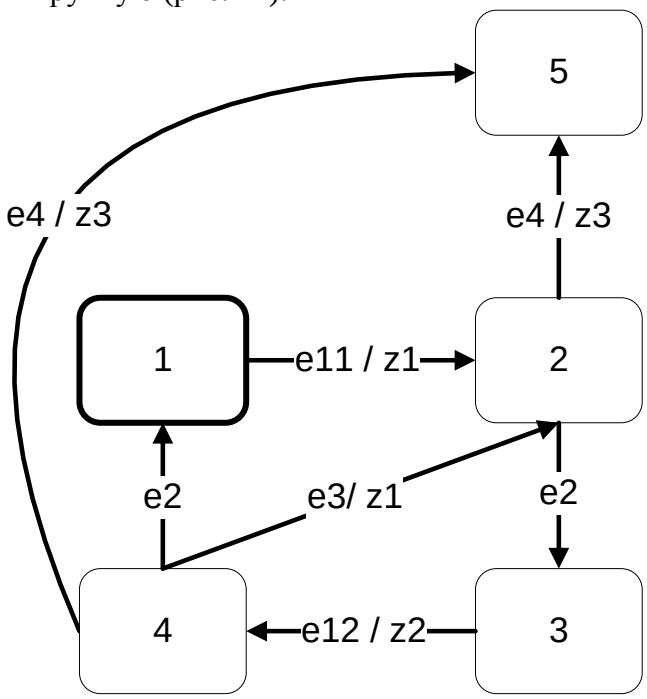


Рис. 21. Граф переходов автомата управления дверьми лифта, построенного на основе тестов и LTL-формул

Начальное состояние имеет номер «1» и выделено жирной рамкой.

3.3.1. Система тестовых примеров

В систему тестов для построения автомата управления дверьми лифта входят девять тестов. Они описывают различные варианты поведения дверей лифта и представлены в табл. 3.

Таблица 3. Тесты для автомата управления дверьми лифта

Тест	Комментарий
Input: *e11, e2, e12, e2* Answer: *z1, z2*	Описывает ситуацию открывания и закрывания дверей лифта.
Input: *e11, e2, e12, e2, e11, e2, e12, e2* Answer: *z1, z2, z1, z2*	Описывает процесс открывания и закрывания дверей лифта дважды.
Input: *e11, e2, e12, e3, e2, e12, e2* Answer: *z1, z2, z1, z2*	Описывает открывание дверей и появления препятствия при закрывании. Дверь закрывается со второго раза.
Input: *e11, e2, e12, e2, e11, e2, e12, e3, e2, e12, e2* Answer: *z1, z2, z1, z2, z1, z2*	Открывание и закрывание дверей лифта, закрывание, снова открывание, и при закрывании появление препятствия.
Input: *e11, e2, e12, e3, e2, e12, e3, e2, e12, e2* Answer: *z1, z2, z1, z2, z1, z2*	Процесс открывания дверей, при закрывании появление препятствия, вторая попытка закрывания и опять препятствие мешает закрыть дверь.
Input: *e11, e4* Answer: *z1, z3*	Описывает процесс возникновения поломки в процессе открывания дверей.
Input: *e11, e2, e12, e4* Answer: *z1, z2, z3*	Описывает процесс возникновения поломки в процессе закрывания дверей.
Input: *e11, e2, e12, e2, e11, e4* Answer: *z1, z2, z1, z3*	Описывает процесс возникновения поломки во время второго открывания дверей лифта.
Input: *e11, e2, e12, e3, e4* Answer: *z1, z2, z1, z3*	Описывает процесс возникновения поломки в момент открывания дверей из-за препятствия.

Совместно с тестами применялись 11 темпоральных свойств. Их описание приведено в табл. 4.

Таблица 4. Темпоральные свойства автомата управления дверьми лифта

Формула	Комментарий
G(wasEvent(e11) => wasAction(z1))	Если было событие e11, то было вызвано действие z1. В терминах модели утверждение можно записать как при обработке события «открыть двери» обязательно будет начато открывание дверей.
G(wasEvent(e12) <=> wasAction(z2))	Событие e12 обрабатывается тогда и только тогда, когда вызывается z2. В терминах модели: при нажатии кнопки закрывания дверей будет начато закрывание, и закрывание дверей может начаться только при нажатии кнопки закрыть двери.
G(wasEvent(e4) <=> wasAction(z3))	Событие e4 обрабатывается тогда и только тогда, когда вызывается z3. В терминах модели: звонок в аварийную службу будет произведен тогда и только тогда, когда лифт сломается.
G(wasEvent(e3) => wasAction(z1))	Если было событие e3, то было вызвано действие z1. Если препятствие мешает закрыть двери, то дверь начнет открываться.
G(wasEvent(e2) => X[wasEvent(e11) || wasEvent(e12)])	Если было событие e2, то следующим обработанным событием будет e11 или e12. В терминах модели: если дверь успешно открылась или закрылась, то следующее обработанное событие может быть только «открыть двери» или «закрыть двери».
G(wasEvent(e11) => X[wasEvent(e4) or wasEvent(e2)])	Если было событие e11, то следующим обработанным событием будет e4 или e2. В терминах модели: если была нажата кнопка «открыть двери», то следующее событие будет либо успешное открывание дверей, либо дверь сломается.
G(wasAction(z1) => X[wasEvent(e2) or wasEvent(e4)])	Если было вызвано действие z1, то следующим обработанным событием будет e2 или e4. В терминах модели: если дверь начала закрываться, то либо она успешно откроется, либо сломается.
G(wasEvent(e12) => X[wasEvent(e2) or wasEvent(e3) or wasEvent(ep.e4)])	Если было событие e12, то следующим обработанным событием будет e2 или e3 или e4. В терминах модели: если нажали кнопку «закрыть двери», то либо двери закроются, либо препятствие помешает закрыть двери, либо лифт сломается.
G(wasAction(z1) => X[U(!wasAction(z1), wasAction(z2) or wasEvent(e4))])	Если было вызвано действие z1, то оно не будет больше вызвано, пока не будет вызвано z2 или событие z4.

G(wasAction(z2) => X[U(!wasAction(z2), wasAction(z1) or wasEvent(e4))])	Если было вызвано действие z2, то оно не будет больше вызвано, пока не будет вызвано z1 или событие z4. В терминах модели: если дверь начала закрываться, то она не будет снова закрываться до тех пор, пока она не будет отрываться или не сломается.
!F(wasEvent(e4) and X(F(wasEvent(e11) || wasEvent(e12) || wasEvent(e2) || wasEvent(e3))))	Не верно, что в будущем будет после события e4 когда либо вызвано e11, e12, e2 или e3. В терминах модели: не верно, что после поломки лифта будут обработаны события «открыть двери», «закрыть двери», успешное открывание или закрывание дверей, препятствие мешает закрыть двери. Или, что то же самое, лифт не может быть починен.

Так же как и при построении автомата управления часами с будильником, при построении автомата, управляющего дверьми лифта, не получается использовать только темпоральные свойства. Но сразу заметим, что и применять только тестовые примеры не получается, так как по ним строится неправильный конечный автомат, о чем будет написано в следующем разделе.

3.3.2. Результаты эксперимента

Для сравнения двух способов построения конечных автоматов (на основе только тестов и на основе тестов и темпоральных свойств) было проведено два варианта экспериментов. Эксперимент каждого типа запускался по 1000 раз. Входные параметры каждого эксперимента были идентичными и эксперименты отличались только наличием или отсутствием *LTL*-формул.

Общие параметры экспериментов представлены в табл. 5.

Таблица 5. Параметры эксперимента построения автомата, управляющего дверьми лифта

Параметр эксперимента	Значение
Размер начальной популяции	2000
Число состояний у автоматов в начальном поколении	6
Ожидаемое число переходов	7
Доля особей, переходящих в следующее поколение. Остальные будут получены с помощью кроссовера	10%
Число поколений до «малой» мутации	70
Число поколений до «большой» мутации	100
Вероятность мутации особи	1%

В результате экспериментов, использующих только тестовые примеры в качестве входных данных, более чем в 99% случаях получался неправильный конечный автомат. Один из таких автоматов представлен на рис. 22, жирной рамкой выделено начальное состояние.

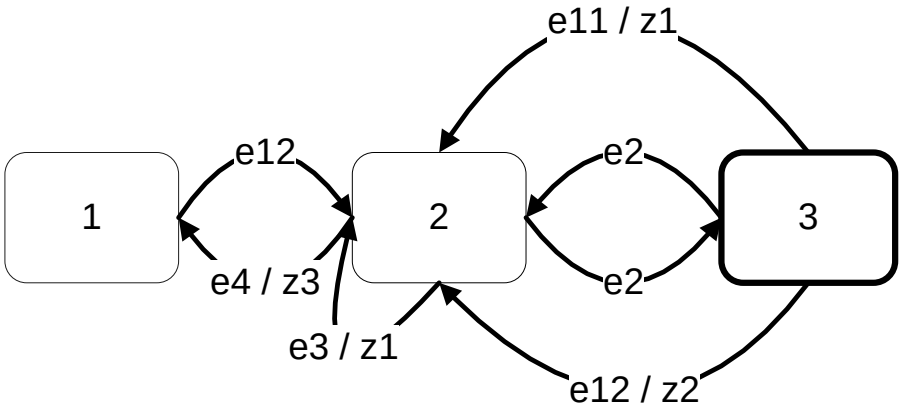


Рис. 22. Граф переходов автомата управления дверьми лифта, построенного только на основе тестов

Ошибка в представленном автомате заключается в том, что после поломки лифта дверь может снова начать закрываться, а затем лифт начнет функционировать как рабочий. Также данный автомат может повторно начать закрывать или открывать двери после закрытия или открытия соответственно.

Заметим, что формально мог построиться и автомат из одного состояния, все переходы которого являлись бы петлями. Такой автомат также проходил бы все тесты, но, естественно, был бы абсолютно неверным.

Указанных проблем можно избежать, применяя верификацию. При построении конечного автомата совместно на основе тестов и *LTL*-формул был построен правильный автомат, изоморфный автомату, изображенному на рис. 21.

Эксперименты производились без разбивки тестов и темпоральных свойств на группы. Функция приспособленности вычислялась по формуле $\mathrm{FF} = \mathrm{FF}_{test} + \mathrm{FF}_{test} \cdot \mathrm{FF}_{LTL} + \frac{1}{10 \cdot M} \cdot (M - \mathrm{cnt})$. В отличие от формулы из раздела 2.2, из вклада тестов ($\mathrm{FF}_{\mathrm{test}}$) удалена надбавка за прохождения всех тестов, так как автомат, проходящий все тесты, может не проходить все формулы. Но, в тоже время, прохождение тестов тоже важно, и автомат, не проходящий тесты, но удовлетворяющий всем формулам имеет нулевую функцию приспособленности. M было взято равным 100, и вклад числа переходов достигал максимума 0.093 на особях, проходящих все тесты и формулы.

Первый вид экспериментов заключался в построении автомата управления дверьми лифта только на основе тестов. Значение функции приспособленности построенного автомата было 1.093. В качестве численной оценки скорости работы алгоритма измерялось число вычислений функции приспособленности при нахождении автомата. Среднее значение вычислений функции приспособленности оказалось равным 7.479×10^4. Минимальное число вычислений – 2.184×10^4. Максимальное число – 2.999×10^5. Среднеквадратичное отклонение – 2.54×10^4. Плотность распределения вероятности числа вычислений функции приспособленности при построении автомата только на основе тестов представлена на рис. 23.

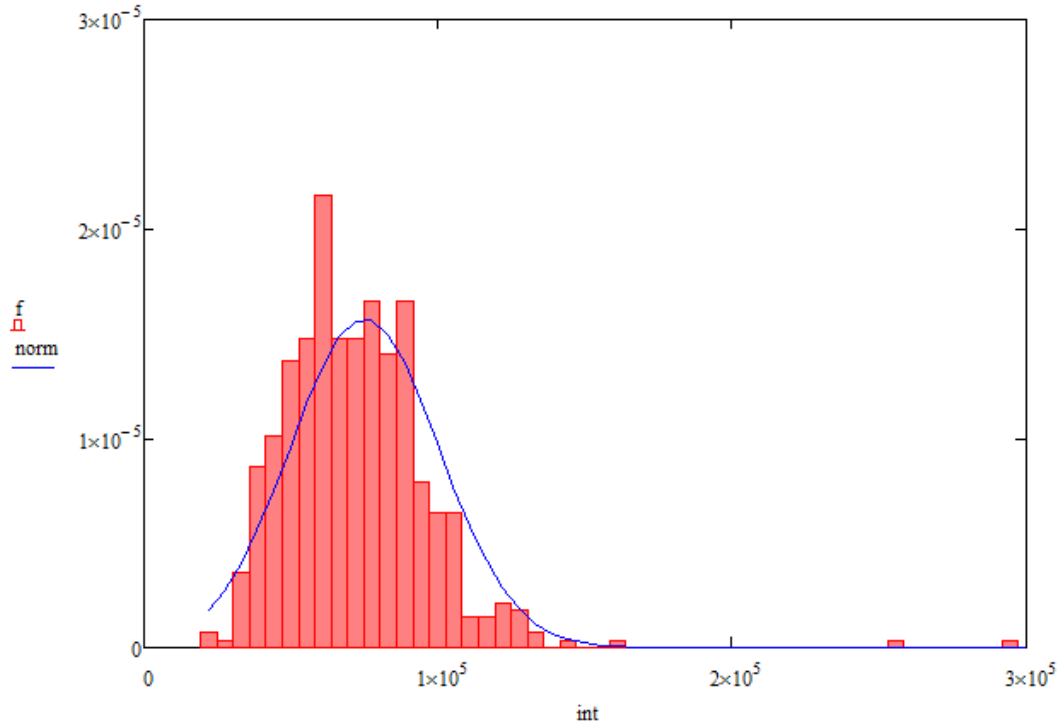


Рис. 23. Плотность распределения вероятности числа вычислений функции приспособленности при построении автомата управления дверьми лифта только на основе тестов

Второй эксперимент заключался в построении автомата, управляющего дверьми лифта на основе тестов и темпоральных свойств. Темпоральные свойства были представлены *LTL*-формулами из предыдущего раздела. Значение функции приспособленности построенного автомата – 2.093. Среднее значение вычислений функции приспособленности оказалось равным 7.246×10^5. Минимальное число вычислений – 7.054×10^4. Максимальное число – 5.492×10^6. Среднеквадратичное отклонение – 7.729×10^5. Плотность распределения вероятности числа вычислений функции приспособленности при построении автомата на основе тестов и *LTL*-формул представлена на рис. 24.

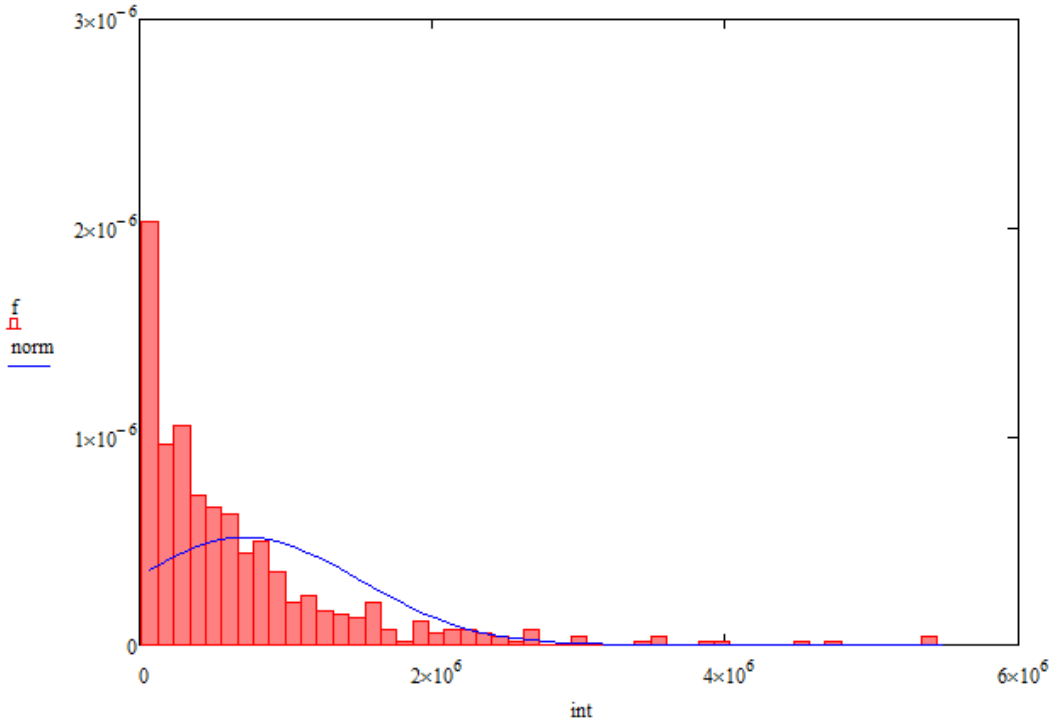


Рис. 24. Плотность распределения вероятности числа вычислений функции приспособленности при построении автомата управления дверьми лифта на основе тестов и LTL-формул

Число вычислений функции приспособленности для экспериментов с *LTL*-формулами оказалось больше практически в 10 раз. Однако, из 1000 построений конечного автомата только на основе тестов, только 9 не содержали ошибок.

Выводы по главе 3

В настоящем разделе были описаны основные моменты программной реализации предлагаемого метода. Разработанный метод дополняет работу [5], поэтому были описаны только сделанные дополнения и изменения.

Также были описаны результаты построения автомата управления часами с будильником и автомата управления дверьми лифта. В обоих случаях верификация хоть и замедляет процесс построения управляющего конечного автомата, но если принять во внимание то, что при построении только на основе тестов процент правильно построенных автоматов меньше 1%, то совместное применение тестов и верификации оправдывает себя.

ЗАКЛЮЧЕНИЕ

В ходе настоящей работы был предложен и реализован метод построения управляющих конечных автоматов на основе тестов и темпоральных свойств. Приведем основные результаты работы:

1. Предложен метод машинного обучения на основе генетических алгоритмов для построения управляющих конечных автоматов Мили на основе тестовых примеров и верификации темпоральных свойств.
2. Предложен метод вычисления функции приспособленности на основе выполнимости или не выполнимости утверждений о конечном автомате.
3. Предложен способ мутации описаний конечных автоматов, учитывающий результаты верификации.
4. Предложен способ скрещивания конечных автоматов, основанный на верификации.
5. Приведена программная реализация этого метода на языке программирования *Java*.
6. Проведено сравнение предложенного метода и метода построения автоматов на основе тестов на задаче построения конечного автомата управления часами с будильником.
7. Показана необходимость проведения верификации на примере построения автомата управления дверьми лифта.

Таким образом, зная спецификацию программы, можно построить управляющий конечный автомат с заранее заданным поведением. Такая программа (конечный автомат) не будет требовать дополнительной верификации и валидации, тем самым полностью соответствуя априорно заданной спецификации.

ИСТОЧНИКИ

1. *Шалыто А. А.* Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.
2. *Angeline P. J., Pollack J.* Evolutionary Module Acquisition // Proceedings of the Second Annual Conference on Evolutionary Programming. 1993. http://www.demo.cs.brandeis.edu/papers/ep93.pdf
3. *Jefferson D., Collins R., Cooper C., Dyer M., Flowers M., Korf R., Taylor C., Wang A.* The Genesys System. 1992. http://www.cs.ucla.edu/~dyer/Papers/AlifeTracker/Alife91Jefferson.html
4. *Chambers L.* Practical Handbook of Genetic Algorithms. Complex Coding Systems. Volume III. CRC Press, 1999.
5. Научно-технический отчет о выполнении Государственного контракта по теме "Разработка методов машинного обучения на основе генетических алгоритмов для построения управляющих конечных автоматов" (этап 1). СПбГУ ИТМО. 2009.
6. *Hoffman L.* Talking Model-Checking Technology // Communications of the ACM. 2008. V. 51. № 7, pp. 110–112.
7. *Кларк Э., Грамберг О., Пелед Д.* Верификация моделей программ: Model Checking. М.: МЦНМО, 2002. 416 с.
8. Разработка технологии верификации управляющих программ со сложным поведением, построенных на основе автоматного подхода. Второй этап. СПбГУ ИТМО, 2007. 105 с. http://is.ifmo.ru/verification/_2007_02_report-verification.pdf
9. *Gerth R., Peled D., Vardi M. Y.,Wolper P.* Simple On-the-fly Automatic Verification of Linear Temporal Logic / Proc. of the 15th Workshop on Protocol Specification, Testing, and Verification. Warsaw. 1995, pp. 3–18.
10. *Courcoubetis C., Vardi M., Wolper P., Yannakakis M.* Memory-Efficient Algorithms for the Verification of Temporal Properties / Formal Methods in System Design. 1992, pp. 275–288.
11. *Егоров К. В., Шалыто А. А.* Методика верификации автоматных программ // Информационно-управляющие системы. СПб: Политехника, 2008, № 5, с. 15–21.
12. *Левенштейн В. И.* Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академии Наук СССР 163.4, с. 845–848.
13. *Gastin P., Oddoux D.* Fast LTL to Büchi Automata Translation / 13th Conference on Computer Aided Verification (CAV'01). 2001, pp. 53–65.
14. LTL2BA project. http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/
15. *Поликарпова Н. И., Шалыто А. А.* Автоматное программирования. СПб: Питер, 2009.
16. *Гуров В. С., Мазим М. А., Нарвский А. С., Шалыто А. А.* UML. SWITCH-технология. Eclipse // Информационно-управляющие системы. 2004. № 6. с. 12–17.