Санкт-Петербургский государственный университет информационных технологий, механики и оптики

М. А. Лукин

Верификация визуальных автоматных программ с использованием инструментального средства *SPIN*

Магистерская диссертация

Научный руководитель – профессор А. А. Шалыто

Санкт-Петербург
2009

Оглавление

Введение

Настоящая работа посвящена созданию методики автоматической верификации визуальных автоматных программ [1, 2]. Практическим приложением данной методики явился инструмент, позволяющий производить верификацию автоматных программ, спроектированных и реализованных при помощи инструментального средства *UniMod* [3]. В данной работе верификация производится с помощью метода *Model Checking* [4 – 8].

Верификация модели программы – это один из основных методов не только поиска ошибок, но и доказательства правильности работы алгоритма. Для того чтобы провести верификацию программы, требуется:

1. Построить формальную модель, приемлемую для инструментальных средств верификации моделей. Обычно при построении модели абстрагируются от несущественных деталей программы в целях упрощения модели. При этом важно не потерять значимые детали. Для автоматных программ этот этап можно произвести автоматически.
2. Сформулировать требования к модели. Обычно их формулируют на языке темпоральной логики. В настоящей работе требования будут формулироваться на языке *LTL* [7 – 11]. Важным вопросом является *полнота* спецификации.
3. Провести верификацию модели. Если модель не соответствует требованиям, пользователь получает трассу с ошибкой (контрпример), которая может помочь найти ошибку в программе. Иногда ошибка происходит в результате некорректного задания модели или неправильной спецификации (требований). В этом случае трасса ошибки помогает устранить ошибку в моделировании или спецификации.

Глава 1. Выбор инструментов. Постановка задачи

1.1. Выбор верификатора

Существует большое число верификаторов, в том числе и с открытыми кодами. В настоящее время самым популярным верификатором является *SPIN* [6 – 9]. Его популярность позволяет надеяться на то, что он не содержит ошибок. На вход этого верификатора подаются модель программы, описанная на языке *Promela* [9], и требования к модели, записанные на языке *LTL*. Верификатор по модели программы строит *модель Крипке* [4 – 6], а по инверсии каждого требования – *автомат Бюхи* [12]. После этого верификатор *SPIN* строит пересечение *модели Крипке* и *автомата Бюхи* «на лету» (не ожидая полного построения структуры *Крипке*), и, если пересечение не пусто, выдается трасса ошибки.

1.2. Постановка задачи

Задача. Требуется разработать метод автоматической верификации визуальных автоматных программ.

Откуда появилась такая задача? Существует большое число верификаторов, в том числе и с открытым кодом. Один из наиболее популярных верификаторов носит название *SPIN*. Традиционно верификация программ с помощью верификатора *SPIN,* происходит следующим образом [4 – 6]:

1. По разработанному алгоритму или программе вручную строится модель на языке *Promela* – входном языке верификатора *SPIN*.
2. Разрабатываются и записываются на языке *LTL* требования (спецификация) к модели, которые верификатор преобразовывает в *автомат Бюхи*, также записанный на языке *Promela*.
3. Проводится верификация построенной модели с помощью верификатора *SPIN*. Верификация построенной модели проходит в несколько этапов:
    - Запускается верификатор с ключом `-a`. В качестве параметра верификатор принимает файл с построенной моделью на языке *Promela*. Верификатор строит программу *pan*.
    - Запускается программа *pan*. Программа преобразует модель на языке *Promela* в *модель Крипке* и верифицирует ее.
    - Запускается верификатор *SPIN* с ключами `-t` (для вывода контрпримера в случае несоответствия модели требованиям) и при необходимости `-p` (с этим ключом выводится более полная информация о контрпримере).

4. Если модель не соответствует требованиям, то проводится анализ контрпримеров. Возможен случай возникновения «ложных опровержений» – ошибка находится не в алгоритме, а в модели. В этом случае требуется изменить модель. Кроме того, модель может быть слишком большой и верификатор не справится с ее верификацией. В этом случае требуется уменьшить модель.

Такой подход имеет следующие недостатки:

- Не гарантируется отсутствие ошибок в программе. Гарантируется лишь правильность алгоритма, на основе которого написана программа.
- Модель приходится строить вручную, а это трудоемкий процесс.

Однако для автоматных программ (в том числе визуальных) модель может быть автоматически построена по самой программе. Таким образом, в этом случае верификация гарантирует правильность работы программы. В этой области уже проводились исследования [13], однако и в этой работе предложенные методы предполагалось ручное построение модели на языке *Promela*.

Таким образом, возникает задача: разработать метод, позволяющий **проводить автоматическую верификацию автоматных программ**. Для решения этой задачи и был выбран верификатор *SPIN*.

Предлагаемый метод состоит из следующих этапов:

1. Генерация модели на языке *Promela* по визуальной автоматной программе.
2. Преобразование проверяемых свойств к виду, который понятен верификатору *SPIN*. Требования должны быть записаны в расширенной предлагаемым методом нотации *LTL*.
3. Верификация сгенерированной модели на языке *Promela* с помощью верификатора.
4. Анализ контрпримера.

Выводы по главе 1

1. Автоматные программы, в отличие от программ общего вида, можно верифицировать автоматически.
2. Поставлена задача: разработать метод автоматической верификации автоматных программ.
3. Для верификации автоматных программ в настоящей работе выбран верификатор *SPIN*.

Глава 2. Основные понятия

2.1. Верификатор *SPIN*

SPIN – эффективный верификатор, который поддерживает верификацию и разработку систем асинхронных процессов. В верификаторе *SPIN* процессы могут взаимодействовать между собой следующими образом:

- с помощью примитивов *rendezvous* [9];
- с помощью асинхронных передач сообщений через буферизованные каналы;
- через общие переменные;
- комбинированным способом.

Кроме собственно верификации, инструментальное средство поддерживает еще режимы интерактивной и случайной эмуляции. При этом запускается интерпретатор модели. Недетерминированные переходы в режиме интерактивной эмуляции выбираются пользователем, а в режиме случайной эмуляции выбираются случайно.

Базовая структура *SPIN* изображена на рис. 1. Типичный режим работы с инструментальным средством [8] – начать со спецификации высокоуровневой модели алгоритма. При этом обычно используют графическую оболочку *Xspin* [9]. После исправления синтаксических ошибок выполняют интерактивную эмуляцию, пока не наступает уверенность, что модель ведет себя, как планировалось. После этого на третьем шаге *SPIN* используется для того, чтобы сгенерировать оптимизированную на лету программу проверки по спецификации высокого уровня. Полученная программа компилируется с возможностью выбора во время компиляции алгоритмов приведения, которые будут использоваться. Если обнаруживаются какие-нибудь контрпримеры к требованиям корректности, то они могут быть возвращены в интерактивный эмулятор и рассмотрены подробно, для того чтобы можно было установить и устранить их причину.

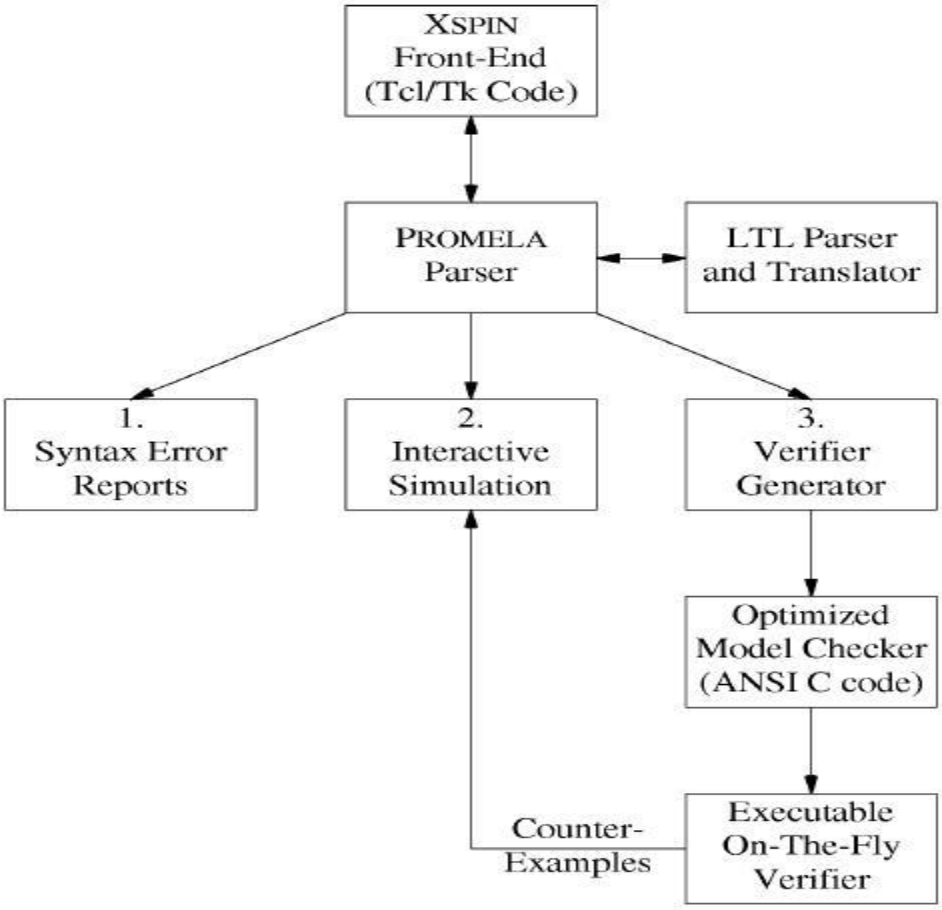

Рис. 1. Базовая структура верификатора *SPIN* [8].

При полностью автоматической верификации автоматных программ работа с верификатором *SPIN* отличается от описанной выше. При условии, что инструментальное средство, которое строит модель, работает корректно, не требуется исправлять синтаксические ошибки и выполнять интерактивную или случайную эмуляцию. Требуется лишь формализовать требования и запустить верификатор.

2.2. Синтаксис языка *Promela*

Синтаксис языка *Promela* напоминает синтаксис языка *C*. Модель на языке *Promela* состоит из следующих элементов:

- объявления типов данных;
- объявления каналов передачи переменных;
- объявления переменных;
- объявления и определения процессов;
- процесса `init`.

Понятие «процесс» в данном случае отдаленно можно рассматривать как процедуру, выполняемую в отдельном потоке. Приведем пример объявления и определения процесса:

```
proctype proc(int a; int b) {
byte b; /* локальная переменная */
/* тело процесса */
}
```

Процессы могут иметь параметры и локальные переменные. Процесс может быть запущен в нескольких экземплярах, если для него используется модификатор `active`. Запускаются процессы с помощью модификатора `Run`.

Язык *Promela* имеет пять базовых типов данных:

- `bit`;
- `bool`;
- `byte`;
- `short`;
- `int`.

Тело процесса состоит из последовательности операторов. Операторы могут быть *выполнимыми* либо *заблокированными*. *Выполнимый* оператор – это такой оператор, который может быть выполнен **немедленно**. *Заблокированный* оператор – оператор, который не может быть выполнен в данный момент. Такой оператор блокирует выполнение процесса до тех пор, пока он не станет *выполнимым*. Например, оператор

```
x < 7
```

может быть выполнен только когда *x* меньше семи. В противном случае он останавливает выполнение процесса до тех пор, пока условие не выполнится. Некоторые операторы, например, оператор присваивания, *выполнимы* всегда.

Язык *Promela* содержит также операторы ветвления и цикла, синтаксис (табл. 1) которых основан на охраняемых командах Дейкстры [14].

Таблица 1. Операторы ветвления и цикла

`if`
`::guard1 -> S1`
`::guard2 -> S2`
`...`
`:: else -> Sk`
`fi`
	`do`
	`::guard1 -> S1`
	`::guard2 -> S2`
	`...`
	`:: else -> Sk`
	`od`

Поясним работу операторов. Если условие $guard_i$ истинно, то выполняется действие S_i. Если одновременно выполняются несколько условий, то происходит недетерминированный выбор одного из них. Если все условия ложны, то выполняется действие S_k, соответствующее *else*. Конструкция *else* может отсутствовать. Тогда в случае, если все условия

ложны, выполнение процесса блокируется до тех пор, пока хотя бы одно из них не начнет выполняться.

2.3. Модель *Крипке*

Пусть *AP* – множество атомарных высказываний. *Модель Крипке* над этим множеством – это четверка $M = (S, S_0, R, L)$:

- S – конечное множество состояний;
- $S_0 \subseteq S$ – множество начальных состояний;
- $R \subseteq S \times S$ – отношение переходов;
- $L : S \rightarrow 2^{AP}$ – функция истинности.

2.4. Язык *LTL*

Для записи требований к модели используются языки темпоральной логики, например, *LTL, CTL, CTL**. Как отмечено выше, в настоящей работе используется язык *LTL* (*Linear-Time Logic*).

Этот язык состоит из множества атомарных высказываний $p_1, p_2, \ldots \in AP$, логических операций ($\neg$, $\wedge$, $\vee$, $\rightarrow$) и темпоральных операторов. Пусть φ – правильно построенная формула. Тогда темпоральные операторы

- $X\varphi$ (в следующем состоянии φ верно – `ne`**`X`**`t`);
- $G\varphi$ (φ верно всегда – **`G`**`lobally`);
- $F\varphi$ (φ когда-нибудь будет верно – **`F`**`inally`);
- $\psi U \varphi$ (ψ будет верно до тех пор, пока не станет верно φ – **`U`**`ntil`)

тоже правильно построенные формулы.

Темпоральные операторы *G* и *F* требуются для упрощения формул, их можно выразить через *U*:

- $F\varphi \equiv 1U\varphi$;
- $G\varphi \equiv \neg F \neg \varphi$.

Формулы *LTL* интерпретируются через исполнение системы переходов в *модели Крипке*. Если все пути из начального состояния удовлетворяют формуле φ, то будем говорить, что поведение системы удовлетворяет формуле φ.

В табл. 2 приведено соответствие стандартного синтаксиса и принятого в верификаторе *SPIN*. Отметим, что *SPIN* не поддерживает оператор *X* `(neXt)`[1], так как *SPIN* генерирует вспомогательные и служебные состояния в *модели Крипке*.

[1] На самом деле можно заставить *SPIN* поддерживать оператор X, скомпилировав *SPIN* с ключом –DNXT. Однако, при этом может нарушиться точность алгоритмов *SPIN*.

Таблица 2. Соответствие стандартного синтаксиса и принятого в верификаторе *SPIN*

[]	G
<>	F
!	¬
U	U
&& или /\	/\
|| или \/	\/
->	→
<->	↔

2.5. Автомат *Бюхи*

Пусть *AP* – множество атомарных высказываний. *Автоматом Бюхи* над алфавитом 2^{AP} называется четверка $A = (Q, q0, \delta, F)$, где

- Q – конечное множество состояний;
- q_0 – начальное состояние;
- $\delta \subseteq Q \times 2^{AP} \times Q$ – функция переходов;
- $F \subseteq Q$ – множество допустимых состояний.

Доказано, что для любой *LTL*-формулы можно построить *автомат Бюхи*, который ее выполняет [15]. Более того, он может строиться автоматически.

Пример. Рассмотрим *LTL*-формулу *G(p U q)*. В нотации *SPIN* эта формула записывается `[](p U q)`. Она обозначает, что всегда гарантировано, что *p* остается верным, по крайней мере, до тех пор, пока не станет верным *q*. Верификатор *SPIN* ее транслирует в конструкцию *never claim:*

```
never {    /* [](p U q) */
T0_init:
  if
  :: ((q)) -> goto accept_S9
  :: ((p)) -> goto T0_init
  fi;
accept_S9:
  if
  :: ((((p)) || ((q)))) -> goto T0_init
  fi;
}
```

Эта конструкция соответствует *автомату Бюхи*, изображенному на (рис. 2). Двойная линия обозначает допустимое состояние.

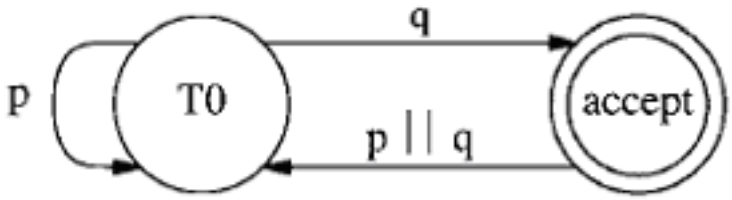

Рис. 2. Автомат *Бюхи* для формулы *G(p U q)*.

С помощью *автомата Бюхи* можно верифицировать *модель Крипке*. С точки зрения верификации автоматных моделей – это наиболее удобный вариант, позволяющий при верификации и спецификации почти полностью ограничиться понятием *конечный автомат*.

Выводы по главе 2

1. Описан верификатор *SPIN* и его входные языки: *Promela* (язык для описания модели) и *LTL* (язык для спецификации требований).
2. Описано соответствие стандартного синтаксиса языка *LTL* и синтаксиса, принятого в верификаторе *SPIN*.
3. Дано понятие об основных структурах, использующихся при верификации: *модель Крипке* и *автомат Бюхи*.

Глава 3. Разработка метода автоматической верификации автоматных программ

3.1. Описание метода верификации

3.1.1. Основная идея метода

Идея метода состоит в следующем:

1. Построить модель автоматной программы на языке *Promela*.
2. Преобразовать *LTL*-формулу с требованиями к автоматной программе к виду, который понятен верификатору *SPIN*.
3. Построенную модель и формулу подать на вход верификатору *SPIN*.
4. Проанализировать отчет, выданный верификатором *SPIN*, и построить контрпример.

3.1.2. Построение модели на языке *Promela*

Реализация (интепретация) автоматной модели (не путать с моделью на языке *Promela*) может изменяться в зависимости от инструментального средства для разработки автоматных программ:

- автоматная программа может быть однопоточной или многопоточной;
- автомат может сначала вызвать вложенные автоматы, а после этого обрабатывать пришедшие сообщения, а может сначала обработать пришедшие сообщения, а если в текущий момент переход сделать невозможно, вызвать вложенные автоматы (так реализовано в инструментальном средстве *Unimod*);
- приоритет в обработке поступившего события может быть у вызывающего автомата, как в *Unimod*, либо у вложенного автомата;
- и т.д.

Опишем метод построения модели. Предполагаем, что автоматная модель реализована, как в инструментальном средстве *Unimod*. Ниже будет изложено, как адаптировать данный метод для других вариантов реализации автоматной модели.

1. Подготовка исходных данных.

1.1. Для каждого автомата *Ai* завести переменную *stateAi*, в которой будет храниться номер текущего состояния. На языке *Promela* это описывается следующим образом:

```
int stateAi;
```

1.2. Для каждой пары автоматов (*Ai*, *Aj*) завести два канала передачи сообщений. На языке *Promela* это описывается следующим образом:

```
chan AiAj = [1] of {byte};
chan AjAi = [1] of {byte};
```

1.3. Завести два канала передачи сообщений для процесса, который эмулирует источник событий:

```
chan epin = [1] of {byte};
chan epout = [1] of {byte};
```

1.4. Завести одну переменную для событий:

```
int lastEvent;
```

1.5. Завести две переменных для выходных воздействий:

```
int o;
int z;
```

1.6. Каждому состоянию присвоить уникальный номер, используя сквозную нумерацию для всех автоматов. Переход в новое состояние с номером *k* будет осуществляться присвоением переменной *stateAi* числа *k*:

```
stateAi = k;
```

1.7. Событие *exx* (*xx* – номер события) на переходе между состояниями описывается в модели следующим кодом:

```
lastEvent = xx;
```

2. Построение

2.1. Для каждого автомата *Ai* создать процесс. На языке *Promela* это записывается следующим образом:

```
active proctype Ai() {
/* тело процесса */
}
```

Модификатор `active` означает, что процесс будет запущен в начальном состоянии модели.

2.2. Для каждого автомата *Ai* в теле созданного процесса выполнить шаги 2.3 – 2.18.

2.3. Определить переменную *x* для приема сообщений от других процессов и переменную *y*, в которой будет храниться номер процесса, от которого получено сообщение:

```
byte x;
byte y;
```

2.4. Для стартового автомата: инициализировать переменные *o* и *z* значением -1:

```
o = -1;
z = -1;
```

2.5. Определить начальное (стартовое) состояние `s`. Присвоить:

```
stateAi = s;
```

2.6. Для всех автоматов, кроме стартового: ждать, пока данному процессу не придет сообщение; в переменную *y* записать номер автомата, приславшего сообщение; на языке *Promela* это записывается следующим образом:

```
if
::A1Ai ? x ->
  y = 1;
::A2Ai ? x ->
  y = 2;
…
::AnAi ? x ->
  y = n;
fi;
```

2.7. Для всех автоматов, кроме стартового, построить цикл:

```
do
::(x == 1) ->
::(x == 9) ->
  break;
od;
```

Здесь сообщение о запуске автомата имеет номер 1, а сообщение о завершении работы системы имеет номер 9.

2.8.Построить цикл:

```
do
::(stateAi == s1) ->
  printf("State Ai l : Имя состояния\n");
::(stateAi == s2) ->
  printf("State 2 : Имя состояния\n");
…
::(stateAi == sk) ->
  printf("State k : Имя состояния\n");
od;
```

Здесь *s1, …, sk* – номера состояний автомата *Ai*.

Инструкция *printf* аналогична соответствующей инструкции из языка *C*. Пометка используется в дальнейшем для восстановления контрпримера. Для всех автоматов, кроме стартового, этот цикл записывается в условие *(x == 1)* цикла, построенного на шаге 2.6.

2.9. Если у состояния есть выходные воздействия, то для каждого выходного воздействия *oi.zj* необходимо записать следующее:

```
printf("Action oi.zj\n");
o = i;
z = j;
o = -1;
z = -1;
```

2.10. Поставить метку (пусть *Ai* и *sj* – текущий автомат и текущее состояние соответственно):

```
Ai_STATE_j:
```

2.11. Оповестить источник событий. Для этого требуется по каналу *epin* отправить число 1. На языке *Promela* это описывается следующим образом:

```
epin ! 1;
```

2.12. Ждать ответ. Если текущий автомат – стартовый, то ждать пока по каналу *epout* от источника событий придет число 1. На языке *Promela* это записывается следующим образом:

```
epout ? 1;
```

Если текущий автомат не является стартовым, то выполнить шаг 2.6.

2.13. Для каждого состояния *sj* найти все возможные переходы *(sj, sl)* из него. К условию *(stateAi == sj)* дописать конструкцию *if*, а для каждого перехода *(sj, sl)* дописать в конструкции *if* следующее:

```
::условие ->
  printf("Going to state Ai l : <Имя состояния>");
  stateAi = sl;
```

Условием может быть наступление события, условие на состояния автоматов и т. д.

2.14. Если в некоторое состояние *sj* вложен автомат *Am*, то дописать в условие *(stateAi == sj)* передачу сообщения на запуск автомата *Am* для процесса этого автомата и ждать завершения и сообщения от поставщика событий (если текущий автомат – стартовый) либо сообщения от другого автомата (если текущий автомат не является стартовым):

```
AiAm ! 1;
if
::AmAi ? x;
::eoput ? 1 ->
  goto Ai_STATE_sj;
fi;
```

либо

```
AiAm ! 1;
if
::A1Ai ? x ->
  y = 1;
::A2Ai ? x ->
  y = 2;
…
::AnAi ? x ->
  y = n;
fi;
```

2.15. Если состояние *st* – конечное, то в условие (*stateAi == st)* дописать инструкцию завершения цикла:

```
break;
```

2.16. Для всех автоматов, кроме стартового: после цикла, построенного на шаге 2.7, дописать отправку сообщения о завершении работы автомата *Ai* процессу автомата, который запустил автомат *Ai*. На языке *Promela* это записывается следующим образом:

```
if
:: y = 1 ->
  A1Ai ! x;
:: y = 2 ->
  A2Ai ! x;
…
:: y = n ->
  AnAi ! x;
fi;
```

2.17. Для всех автоматов, кроме стартового, ждать, пока данному процессу не придет сообщение. Выполнить шаг 2.6.

2.18. Для стартового автомата. После цикла, построенного на шаге 2.7, отправить всем остальным процессам сообщение о завершении работы:

```
AiA1 ! 9;
AiA2 ! 9;
…
AiAn ! 9;
epin ! 9;
```

2.19. Дописать в модель источник событий.

```
active proctype eventProvider() {
  byte x;
  do
  ::epin ? 1 ->
    if
    ::lastEvent = 0;
    …
    ::lastEvent = M;
    ::lastEvent = -1;
    fi;
    epout ! 1;
  ::epin ? 9 ->
    lastEvent = -1;
    epout ! 9;
    break;
  od;
```

Здесь все события имеют номера от 0 до *M*. Минус единица означает, что никакого события не произошло.

2.20. Дописать в модель требования (проверяемые свойства), преобразованные с помощью верификатора *SPIN* с языка *LTL* на язык *Promela*.

Предположим, что система автоматов многопоточная. Тогда для каждого автомата, который запускается в отдельном потоке, модель на языке *Promela* строится так же, как для стартового автомата, за исключением шага 2.6. Кроме того, на шаге 2.14 (вызов вложенного автомата) требуется убрать ожидание сообщения от других автоматов.

Для построения модели на языке *Promela* варианта реализации автоматной модели, когда приоритет обработки события отдается вложенному автомату, требуется для всех вложенных автоматов добавить к ожиданию сообщения от других автоматов ожидание сообщения от источника событий

3.1.3. Преобразование *LTL*-формул

Нотация верификатора *SPIN* для языка *LTL* предполагает, что все элементарные высказывания записаны в виде некоторых идентификаторов, которые расшифровываются отдельно [16]. Излагаемый метод расширяет указанную нотацию *SPIN*, добавляя возможность записи элементарных высказываний в формуле. В качестве элементарного высказывания может использоваться любое выражение на языке *Promela*. Элементарное высказывание требуется записывать в фигурных скобках.

Опишем алгоритм преобразования формулы, записанной в расширенной нотации, к формуле в нотации верификатора *SPIN*.

1. Алгоритму на вход подается строка с формулой.

2. Вводим счетчик элементарных высказываний.
3. Пока не кончилась строка, идем по строке в поиске открывающих фигурных скобок.
4. Если нашли открывающую фигурную скобку, то увеличиваем счетчик на единицу и идем по строке дальше в поиске закрывающей фигурной скобки.
5. Если закрывающая фигурная скобка не была найдена до конца строки или до момента появления новой открывающей фигурной скобки, то *LTL*-формула неправильная. Прекращаем работу.
6. Если закрывающая фигурная скобка была найдена, то заменим элементарное высказывание, находящееся между фигурными скобками на идентификатор `pk`, который состоит из символа `p` и числа `k`, которое показывает счетчик элементарных высказываний, и запишем в модель следующий код:

```
#define pk (элементарное высказывание)
```

Переходим к шагу 3.

7. Если строка закончилась, то завершаем работу.

В таком виде *LTL*-формула может быть обработана верификатором *SPIN* (при условии, что темпоральные операторы записаны в нотации верификатора *SPIN* [16]).

Заметим, что данный алгоритм не проверяет правильность написания элементарных высказываний, оставляя эту проверку верификатору *SPIN*.

3.1.4. Построение контрпримера

Верификатор *SPIN* в качестве внутреннего представления модели строит *модель Крипке* и производит верификацию построенной модели. В случае несоответствия модели проверяемым свойствам верификатор выводит контрпример как путь в модели на языке *Promela*, поданной ему на вход. В модель на языке *Promela* встроены пометки, в которых содержится текущее состояние автомата. Таким образом, в сгенерированном верификатором отчете содержится вся информация о контрпримере.

3.2. Формализация типовых требований к автоматным моделям

Рассмотрим типовые требования, которые можно применять ко всем автоматным моделям [17].

Завершение работы. В рамках автоматной модели в инструментальном средстве *UniMod* это требование может быть записано на языке темпоральной логики следующим образом:

$$F(state == end\_state),$$

где *state* – состояние главного автомата, а *end_state* – конечное состояние. Эта формула обозначает, что главный автомат когда-нибудь попадет в конечное состояние.

Прогресс. Это требование состоит в том, что в любой момент выполнения программы автомат может прийти в состояние *s* или произойдет событие *exx*. На языке линейной темпоральной логики оно записывается так:

$$GF(state == s) \text{ или } GF(lastEvent == exx).$$

Постоянство. Это требование состоит в том, что некоторое свойство *p* с некоторого момента выполняется всегда. Это свойство применимо не только к автоматным программам, а к любой модели. Его запись на языке *LTL* имеет вид:

$$FG(p).$$

В данном разделе не описаны такие требования как *корректное завершение работы*, так как темпоральная формула для них зависит от конкретной модели.

3.3. Практическая реализация метода

3.3.1. Общее описание инструментального средства *Converter 2.0*, разработанного в настоящей работе

Этот инструмент является практической реализацией предложенного выше метода верификации автоматных программ и позволяет *проводить полностью автоматическую верификацию автоматных программ*.

По автоматной программе *Converter* создает модель, в которой отброшены несущественные детали. *LTL*-формула преобразовывается в пригодный для верификатора *SPIN* вид.

При построении модели используются:

- графы переходов автоматов;
- взаимодействие автоматов по вложенности;
- выходные воздействия, обозначаемые, как *z* с соответствующими индексами;
- события на переходах.

Построенная модель абстрагируется от следующих сущностей:

- входных переменных, обозначаемых, как *x* с соответствующими индексами;

- выходных переменных, обозначаемых, как *y* с соответствующими индексами.

Такая абстракция позволяет генерировать более простые модели, обеспечивая возможность верификации программ большей размерности. Вместе с тем, более подробная модель позволяет более точно верифицировать программы.

Инструментальное средство принимает следующие входные параметры:

- путь к *xml*-описанию автоматной программы;
- имя файла, в который инструментальное средство сохранит построенную модель;
- *LTL*-формулу;
- имя файла, в который инструментальное средство сохранит отчет о проведенной верификации.

Если *LTL*-формула не была задана, то верификация проводиться не будет. В этом случае *Converter* только построит модель автоматной программы.

Если не был задан какой-то из остальных параметров, то инструментальное средство использует значение по умолчанию. В частности, если не был задан путь к *xml*-описанию автоматной программы, *Converter* будет использовать в качестве входного файла `A1.xml`.

Для построения *LTL*-формул используются следующие переменные:

- *stateAi* для проверки условий на состояние автомата *Ai*;
- *lastEvent* для проверки условий на произошедшие события;
- Две переменные *o* и *z* для проверки условий на выходные воздействия. Переменная *o* используется для обозначения номера объекта управления, переменная *z* используется для обозначения индекса выходного воздействия.

3.3.2. Архитектура инструментального средства *Converter 2.0*

3.3.1.1. Первичное отображение

В первичном отображении представлены диаграмма классов (рис. 3) и представление декомпозиции на модули в виде диаграммы компонентов (рис. 4).

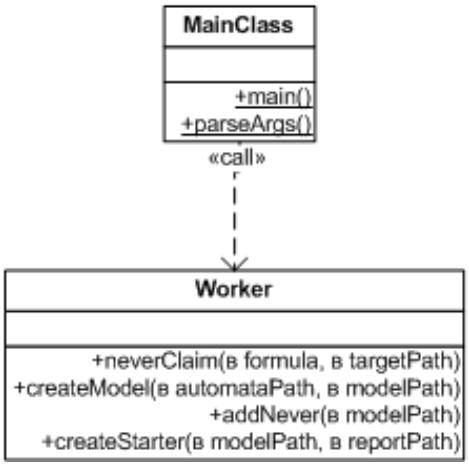

Рис. 3. Диаграмма классов

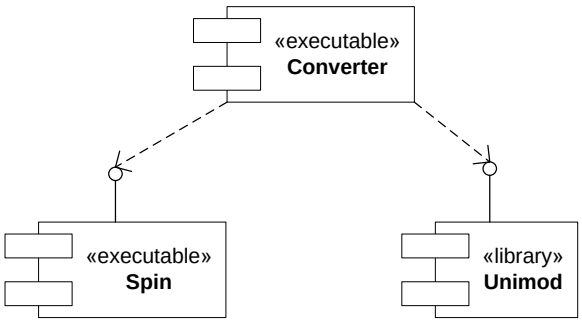

Рис. 4. Диаграмма компонентов

3.3.1.2. Каталог элементов

Каталог элементов приведен в табл. 3.

Таблица 3. Каталог элементов

Элемент	Описание
1. Компонент *Unimod*	Библиотека инструментального средства *Unimod*. Используется для чтения автоматной программы из *XML*-файла.
2. Компонент *Spin*	Инструментальное средство *SPIN* и программа *pan*, которую строит *SPIN* во время работы. Производит верификацию модели.
3. Компонент *Converter*	Запускается в пакетном режиме. Решает следующие задачи: • строит из автоматной программы модель; • преобразовывает *LTL*-формулу; • подает построенную модель и *LTL*-формулу на вход компоненту *Spin*. • собирает отчет о проведенной верификации.
3.1. Класс *MainClass*	Содержит в себе пользовательский интерфейс.
3.1.1. Метод *main*	Точка входа в программу *Converter*.
3.1.2. Метод *parseArgs*	Производит разбор входных параметров.
3.2. Класс *Worker*	Решает следующие задачи: • строит из автоматной программы модель; • преобразовывает *LTL*-формулу; • подает построенную модель и *LTL*-формулу на вход компоненту *Spin*; • собирает отчет о проведенной верификации.
3.2.1. Метод *neverClaim*	Преобразовывает *LTL*-формулу в формат верификатора *SPIN* и запускает парсер *LTL*-формул верификатора *SPIN*. В результате работы получается конструкция *never claim*.
3.2.2. Метод *createModel*	Создает модель автоматной программы на языке *Promela*.
3.2.3. Метод *addNever*	Копирует конструкцию *never claim*, сгенерированную при помощи метода *neverClaim* в конец файла с построенной моделью.
3.2.4. Метод *createStarter*	Запускает инструментальное средство *SPIN* в режиме верификации.

3.3.1.3. Пользовательский интерфейс

Инструментальное средство *Converter 2.0* запускается в пакетном режиме и имеет консольный пользовательский интерфейс. Командная строка для запуска инструментального средства выглядит следующим образом:

```
Converter [список ключей].
```

Входные параметры задаются при помощи следующих ключей:

- `-s <путь к файлу>` – задать путь к *XML*-описанию автоматной программы. Если данный параметр отсутствует, то будет использоваться значение по умолчанию `A1.xml`.
- `-m <path>` – задать путь к файлу, в который будет сохранена построенная модель. Если данный параметр отсутствует, то будет использоваться значение по умолчанию `model.ltl`.
- `-r <path>` – задать путь к файлу, в который будет записан отчет о проведенной верификации. Если данный параметр отсутствует, то будет использоваться значение по умолчанию `report.txt`.
- `-f <formula>` – задать *LTL*-формулу. Если параметр отсутствует, то верификация не проводится.
- `-h` – вывести справку и завершить работу.

3.3.3. Описание работы инструментального средства

Автоматическая верификация автоматных программ состоит из нескольких этапов (рис. 5).

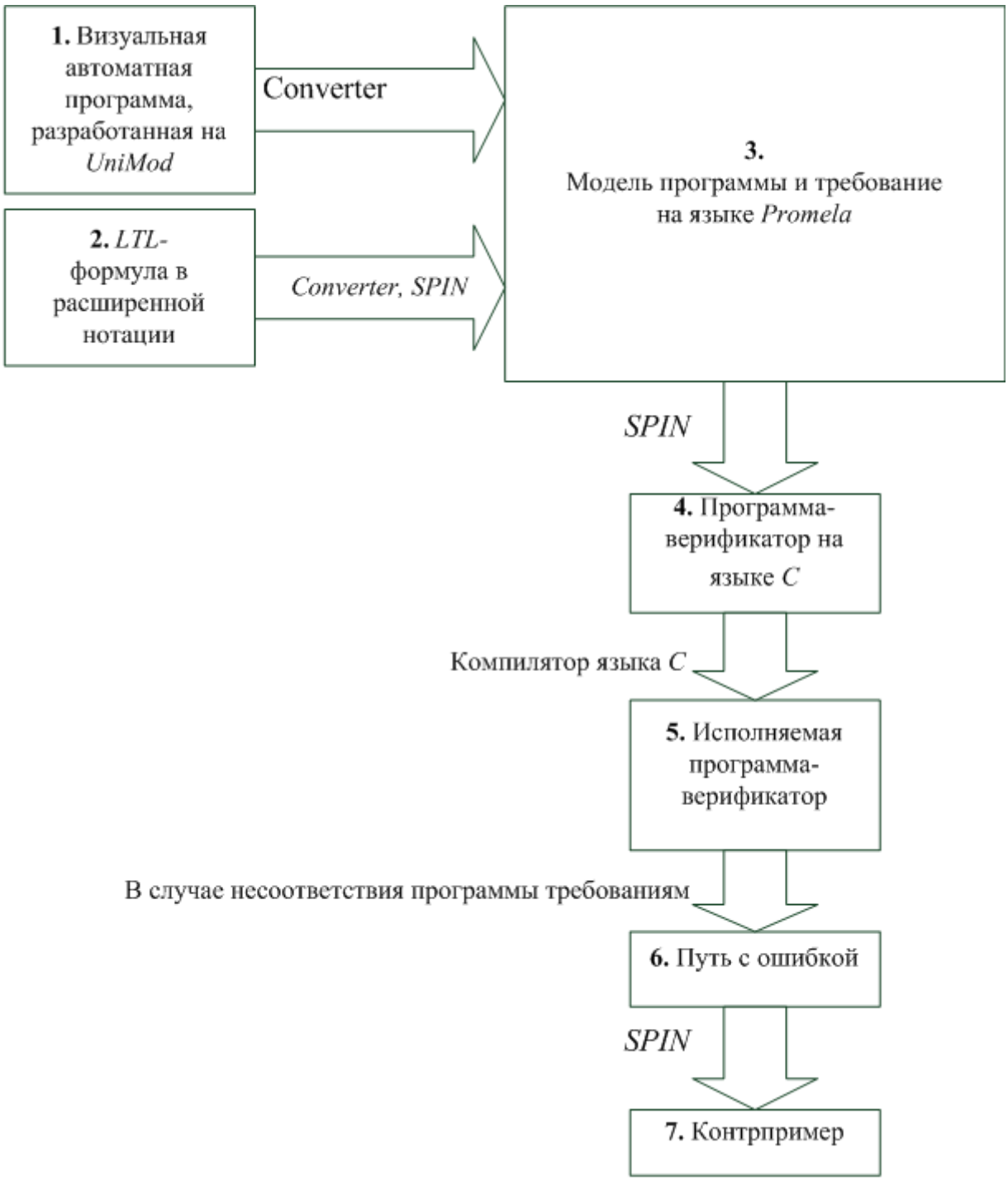

Рис. 5. Диаграмма, поясняющая работу инструмента *Converter*

1. *Converter* принимает на вход визуальную автоматную программу, разработанную при помощи инструментального средства *UniMod* и сохраненную в формате *XML*, и требования к ней, записанные на языке *LTL* в расширенной нотации (рис. 5, переходы от 1 и 2). **Важно: верификатору на вход подаются не требования, а их отрицание.**
2. При помощи *UniMod* из *XML*-файла получается автоматная модель на языке *Java* (рис. 6) – переход 1 – 3 на рис. 5.

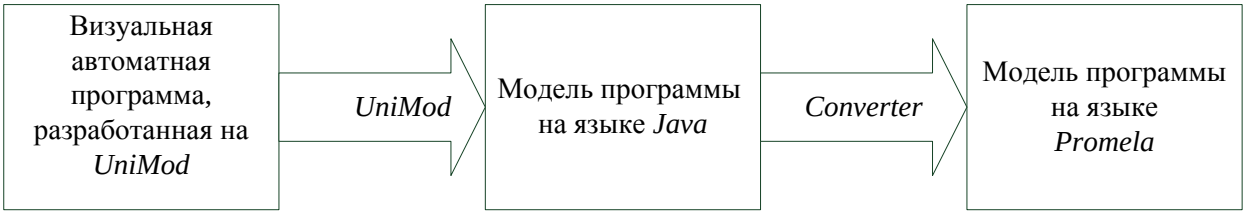

Рис. 6. Взаимодействие с *UniMod*

3. *Converter* транслирует автоматную модель с языка *Java* на язык *Promela* (рис. 6) – переход 1 – 3 на рис. 5.
4. *Converter* преобразует *LTL*-формулу в нотацию верификатора *SPIN* (рис. 7). Это преобразование выполняется следующим образом (это также переход 2 – 3 на рис. 5):

Рис. 7. Взаимодействие с *UniMod*

- Все элементарные высказывания должны быть записаны в фигурных скобках.
- Все элементарные высказывания должны удовлетворять синтаксису языка *Promela*.
- Каждому элементарному высказыванию присваивается идентификатор *pk*, где *k* – порядковый номер элементарного высказывания в формуле.
- Для каждого элементарного высказывания *Converter* генерирует макрос на языке *Promela*, закрепляющий идентификатор за элементарным высказыванием. Если *Converter* распознает формулу как неправильную, то он выводит в консоль сообщение «Wrong formula».
- Идентификаторы событий *exx*, где *xx* – номер события, преобразовываются в число *xx*.
- Пример. Формула (элементарное высказывание)
  `{lastEvent == e13}`
  преобразовывается в строку на *Promela*:
  `#define pk (lastEvent == 13)`,
  где *k* – номер элементарного высказывания.

5. *Converter* запускает верификатор *SPIN* в режиме генерации конструкции *never claim* (с помощью команды `spin -f <формула>`). Верификатор по *LTL*-формуле генерирует конструкцию n*ever claim*,

представляющую собой *автомат Бюхи*, записанный на языке *Promela* (рис. 7). Это также переход 2 – 3 на рис. 5.

6. После того, как на языке *Promela* описаны модель и требование к ней, *Converter* запускает *SPIN* на верификацию (командой `spin -a <Модель>`). Верификатор *SPIN* генерирует файл `pan.c`, представляющий собой программу-верификатор на языке *C* (переход 3 – 4 на рис. 5).
7. После компиляции файла `pan.c` получается программа-верификатор для данной конкретной модели с заданным требованием. Инструментальное средство *Conveter* запускает программу *pan* с параметрами `-a -n -I`. Ключ `-a` требуется для подключения циклов при верификации. Ключ `-n` исключает из отчета лишние подробности. Ключ `-I` требуется для того, чтобы верификатор искал самый короткий контрпример. Программа *pan* выполняет верификацию построенной модели. При обнаружении ошибок программа *pan* выдает *trail*-файл, в котором описана трасса ошибки в формате, понятном верификатору *SPIN* (переходы 4 – 6 на рис. 5). Кроме того, программа-верификатор *pan* выводит отчет, в котором содержится краткая информация об ошибках (переход 5 – 6 на рис. 5), использованной памяти, версия *SPIN* и т. д.
8. По команде `spin -t <Модель>` верификатор выводит отчет, содержащий контрпример. *Converter* собирает воедино отчет, созданный *SPIN*, и отчет, созданный программой *pan* (переход 6 – 7 на рис. 5).

3.3.4. Обратное преобразование контрпримера

Инструментальное средство *Converter 2.0* на выходе выдает текстовый файл с отчетом, в котором содержатся полные сведения о проведенной верификации. В начале отчета содержатся общие сведения о сгенерированной *модели Крипке*, о режиме работы верификатора, об использованной памяти, о числе ошибок в модели (если они есть) и т. д. Если модель не удовлетворяет *LTL*-формуле, то в отчете будет следующая запись:

```
pan: claim violated!
```

Если модель не содержит ошибок, то в отчете будет строка следующего вида:

```
State-vector 24 byte, depth reached 0, errors: 0.
```

Верификатор *SPIN* выдает контрпример в виде пути в модели, описанной на языке *Promela*. Изложим, как из него вернуться к автоматной программе.

Строка отчета

```
StateAk s : <имя состояния>
```

(*stateAk* – состояние автомата *Ak*, *s* – номер состояния) обозначает вход автомата *Ak* в состояние *s*.

Строка отчета

```
Going to StateAk s : <имя состояния>
```

(*stateAk* – состояние автомата *Ak*, *s* – номер состояния) обозначает переход автомата *Ak* из текущего состояния в состояние *s*.

Строка отчета

```
Event = exx
```

(*exx* – некоторое событие) обозначает, что был совершен переход по событию *exx*.

Строка отчета

```
Action oi.zj
```

обозначает, что было вызвано выходное воздействие *zj* объекта управления *oi*.

Таким образом, строится путь в автоматной модели из файла отчета, выданного инструментом *Converter*.

3.4. Характеристики метода

Проанализируем скорость работы алгоритма создания по визуальной автоматной программе модели на языке *Promela*.

Введем обозначения. Обозначим переменной *a* число автоматов. Обозначим переменной *i* номер текущего автомата. Обозначим переменной *j* номер текущего состояния в автомате.

Для каждого автомата с номером *i* обозначим переменной s_i число состояний в этом автомате, а переменной *s* – общее число состояний в автоматной программе. Для каждого состояния *j* в автомате *i* обозначим переменной p_{ij} число переходов, которые исходят из этого состояния, переменной p_i – число переходов в автомате *i*, а переменной *p* – общее число переходов между состояниями в программе. Для каждого перехода с номером *k* обозначим переменной e_{ijk} число событий на данном переходе (e_{ijk} равно либо нулю, либо единице). Во всех состояниях *j* переменной e_{ij} обозначим число событий на всех переходах, которые исходят из этого состояния. Переменная e_i обозначает число событий на всех переходах в автомате *i*, а *e* – общее число событий на переходах в программе. Таким образом, каждое событие будет посчитано столько раз, на скольких переходах оно используется.

Для каждого автомата *i* согласно предлагаемому алгоритму генерируется отдельная функция за константное время. В каждое состояние *j* автомата алгоритм построения модели заходит ровно один раз. Алгоритм генерирует общий код для состояния за время $\Theta(1)$ и обрабатывает все переходы, которые ведут из этого состояния. Генерация кода для каждого

перехода *k* из состояния занимает время Θ*(1)* плюс время, требуемое на обработку всех событий. Для обработки одного события требуется время Θ*(1)*. Таким образом, для обработки одного перехода требуется время Θ(e_{ijk}). Тогда на обработку одного состояния требуется время $\Theta(\sum_k (1+e_{ijk})) = \Theta(p_{ij}+e_{ij})$. Исходя из этого, время, требуемое на обработку одного автомата, пропорционально $\sum_j (1+\Theta(p_{ij}+e_{ij})) = \Theta(s_i+p_i+e_i)$, а время, требуемое на обработку всей программы, пропорционально $\sum_i (1+\Theta(s_i+p_i+e_i)) = \Theta(a+s+p+e)$.

Таким образом, время работы алгоритма создания для автоматной визуальной программы модели на языке *Promela* линейно:

- по числу автоматов;
- по общему числу состояний в автоматной программе;
- по общему числу переходов между состояниями в программе;
- по числу событий на переходах.

Определим время работы алгоритма преобразования *LTL*-формул. Алгоритм проходит по всей формуле один раз и для каждой последовательности символов в фигурных скобках *{expression}* с номером *k* делает следующее:

- в модель пишет определение `#define pk (expression);`
- заменяет выражение *{expression}* на идентификатор *pk* в формуле.

Каждое выражение *{expression}* обрабатывается за время, пропорциональное его длине. Следовательно, время, которое требуется для работы алгоритма преобразования *LTL*-формул, пропорционально длине строки с формулой.

Таким образом, метод обеспечивает линейный рост модели на языке *Promela*.

3.5 Сравнение с существующими решениями

В этом разделе приведено сравнение с известными методами автоматической верификации автоматных программ при помощи *LTL*-свойств [18 – 20]:

1. Метод верификации автоматных программ при помощи инструментального средства *SPIN*, разработанный автором настоящей работы в бакалаврской работе. Практической реализацией разработанного метода является инструментальное средство *Converter 0.50*.
2. Метод верификации автоматных программ при помощи инструментального средства *Bogor*, разработанный Б. Яминовым. Практической реализацией этого метода является инструментальное средство *UniMod.Verifier*.

3. Метод верификации автоматных программ, разработанный К. Егоровым. Практической реализацией разработанного метода является инструментальное средство *Automata Verificator*.

3.5.1. Многопоточность

С возрастанием числа ядер у процессоров все большую актуальность приобретают многопоточные программы. Поэтому важно уметь верифицировать автоматные программы этого класса.

Все три вышеприведенных метода позволяют верифицировать автоматные программы, исполняющиеся в одном потоке.

Метод верификации автоматных программ, описанный в настоящей работе, позволяет верифицировать и многопоточные автоматные программы.

3.5.2. Работа с большими автоматами

Для тестирования производительности были программно сгенерированы автоматные программы, имеющие следующую структуру:

- Автомат имеет *N* состояний *s1,…,sN*.
- Из каждого состояния *si* выходит пять переходов в следующие пять состояний, если эти состояния существуют. Таким образом, из состояния *sN-1* выходит один переход, а из состояния *sN* – ни одного.

Инструментальное средство *Converter 0.50* позволяет верифицировать автоматы с не более чем 214 состояниями [22] из-за ограничений верификатора *SPIN*.

Инструментальное средство *UniMod.Verifier* верифицирует автомат из 1000 состояний более, чем за 10 минут.

Инструментальное средство *Automata Verificator* верифицирует более 500 000 состояний за три минуты.

Инструментальное средство *Converter 2.0* позволяет верифицировать автоматы с более чем 4900 состояниями. Автоматы с 4900 состояниями верифицируются примерно за три с половиной минуты.

3.5.3. Работа со сложной автоматной структурой

Рассмотрим автоматную структуру, описанную в работе [21]. Данная структура состоит из двух автоматов: *A1* и *A2*. На рис. 8, 9 приведены диаграммы состояний этих автоматов.

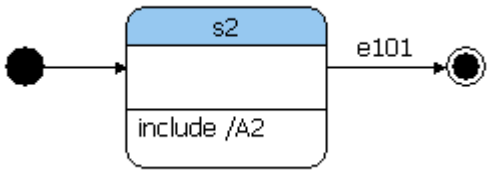

Рис. 8. Диаграмма состояний автомата *A1*

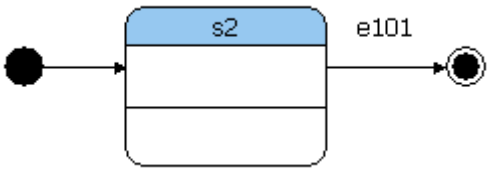

Рис. 9. Диаграмма состояний автомата *A2*

Событие *e101* происходит по таймеру – через секунду после запуска этой автоматной программы в инструментальном средстве *Unimod*. При запуске программы инициализируется автомат *A1*. Он переходит в состояние *s2*, после этого по событию *e101* автомат *A1* переходит в конечное состояние, и программа завершает работу.

Проверим с помощью верификаторов следующее утверждение: «*Автомат A2 никогда не попадет в состояние s3*». Ожидаемый результат: сообщение о том, что ошибок в модели нет.

Приведем номера, которые присвоили состояниям автоматов верификаторы *Converter 0.50* и *Converter 2.0*:

Автомат *A1*:

- *s1*: 1,
- *s2*: 2,
- *s3*: 3.

Автомат *A2*:

- *s1*: 5
- *s2*: 6,
- *s3*: 7.

Таким образом, для верификаторов *Converter* данное свойство формулируется следующим образом:

```
<>{stateA2 == 7}.
```

Для верификатора *UniMod.Verifier* данное свойство записывается следующим образом:

```
G ! (A2 in state s3).
```

Для верификатора *Automata Verificator* данное свойство формулируется так:

```
G!(isInState(A2, A2[\"s3\"])).
```

Верификатор *Converter 0.50* выдает следующий контрпример:

```
  2:  proc  0 (:init:) line  67 "model.ltl" (state 1)
          [(run Model())]
  4:  proc  1 (Model) line  11 "model.ltl" (state 1)
          [stateA1 = 1]
  6:  proc  1 (Model) line  13 "model.ltl" (state 2)
          [((stateA1==1))]
              State A1 1 : s1
  8:  proc  1 (Model) line  14 "model.ltl" (state 3)
          [printf('State A1 1 : s1\\n')]
 10:  proc  1 (Model) line  16 "model.ltl" (state 4)
          [stateA1 = 2]
              Going to state A1 2 : s2
 12:  proc  1 (Model) line  17 "model.ltl" (state 5)
          [printf('Going to state A1 2 : s2\\n')]
 14:  proc  1 (Model) line  19 "model.ltl" (state 8)
          [((stateA1==2))]
              State A1 2 : s2
 16:  proc  1 (Model) line  20 "model.ltl" (state 9)
          [printf('State A1 2 : s2\\n')]
              Calling automaton A2
 16:  proc  1 (Model) line  21 "model.ltl" (state 10)
          [printf('Calling automaton A2\\n')]
 18:  proc  1 (Model) line  41 "model.ltl" (state 11)
          [stateA2 = 5]
 20:  proc  1 (Model) line  43 "model.ltl" (state 12)
          [((stateA2==5))]
              State A2 5 : s1
 22:  proc  1 (Model) line  44 "model.ltl" (state 13)
          [printf('State A2 5 : s1\\n')]
 24:  proc  1 (Model) line  46 "model.ltl" (state 14)
          [stateA2 = 6]
              Going to state A2 6 : s2
 26:  proc  1 (Model) line  47 "model.ltl" (state 15)
          [printf('Going to state A2 6 : s2\\n')]
 28:  proc  1 (Model) line  49 "model.ltl" (state 18)
          [((stateA2==6))]
              State A2 6 : s2
 30:  proc  1 (Model) line  50 "model.ltl" (state 19)
          [printf('State A2 6 : s2\\n')]
 32:  proc  1 (Model) line  52 "model.ltl" (state 20)
          [stateA2 = 7]
Never claim moves to line 72       [((stateA2==7))]
              Going to state A2 7 : s3
 34:  proc  1 (Model) line  53 "model.ltl" (state 21)
          [printf('Going to state A2 7 : s3\\n')]
```

Таким образом, *Converter 0.50* нашел следующий контрпример: автомат *A1* переходит в состояние *s2*, вызывает автомат *A2*, автомат *A2* переходит в состояние *s2*. После этого происходит событие *e101* и автомат *A2* переходит в состояние *s3*.

Верификатор *UniMod.Verifier* выдает результат, что верификация прошла успешно.

Верификатор *Automata Verificator* выдает следующий контрпример:

```
DFS 2 stack:
-->["<"s3", "s3">", 0, 0]-->["<"s3", "s3">", 0, 0]
DFS 1 stack:
-->["<"s1", "s1">", 1, 0]-->["<"s2", "s1">", 1, 0]--
        >["<"s2", "s2">", 1, 0]
-->["<"s2", "s3">", 0, 0]-->["<"s3", "s3">", 0, 0]
```

Automata Verificator нашел точно такой же контрпример, что и *Converter 0.50*.

Верификатор *Converter 2.0* выдает результат о том, что ошибок нет.

В результате, правильный ответ выдали только два верификатора: *UniMod.Verifier* и *Converter 2.0*.

Таким образом, разработанное в настоящей работе инструментальное средство *Converter 2.0* уступает в производительности только *Automata Verificator*. Это, по-видимому, связано с тем, что в *Automata Verificator* *модель Крипке* строится с учетом того, что эта модель строится для автоматной системы, а верификаторы *SPIN* и *Bogor* строят *модель Крипке* без учета специфики задачи.

Предлагаемый метод отличается от остальных методов своей универсальностью. Каждый из рассмотренных выше методов разрабатывался для конкретного вида интерпретации автоматной модели. При этом, интерпретация автоматной модели в методах [18, 20] не совпадает с интерпретацией инструментального средства *Unimod*.

3.6. Сравнение с ручными методами верификации

Под ручной верификацией автор понимает процесс верификации, при котором построение модели осуществляется вручную.

При ручном построении модели невозможно избежать человеческого фактора, а, следовательно, нельзя гарантировать, что построенная модель точно соответствует исходной программе. При автоматическом построении модели при условии, что инструментальное средство работает корректно, гарантируется соответствие построенной модели и исходной программы. Таким образом, основное отличие состоит в том, что при ручной верификации не гарантируется, что если построенная модель соответствует спецификации, то и программа, по которой построена модель, соответствует этой спецификации.

Следовательно, при помощи ручной верификации можно проверить лишь правильность алгоритма, но не программы.

Кроме того, построение модели – трудоемкий процесс. Построение подробной модели сравнимо по стоимости с разработкой программы.

Однако ручной метод построения модели – более гибкий. В каждом конкретном случае можно выбрать свой уровень абстракции.

Выводы по главе 3

1. Данная работа является продолжением бакалаврской работы [18]. Метод, разработанный в бакалаврской работе, обладал существенным недостатком: жестким ограничением на размер автомата [22]. В настоящей работе был разработан новый алгоритм построения модели, и поэтому ограничение на размер автомата возросло более чем на порядок.
2. Разработан универсальный метод, позволяющий проводить верификацию для различных вариантов интерпретации автоматной модели.

Глава 4. Верификация банкомата

4.1. Описание банкомата

Клиентская программа запускается и предлагает пользователю выполнять различные операции с его кредитной картой.

Первое, что требуется от пользователя – вставить карту. В этом банкомате этот процесс эмулируется вводом номера карты. В реальном банкомате номер карты считывается с нее автоматически.

Далее пользователь вводит *pin*-код. Если на сервере не найдется запись о счете, с введенным номером и *pin*-кодом, то работа с этой картой прекращается. Если же *pin*-код и номер счета были введены правильно, то пользователю предлагается выполнить одну из следующих операций:

- «Забрать карту» – возврат карты. Все текущие операции отменяются, а карта возвращается на руки пользователю.
- «Баланс» – отображает текущий остаток на счете, выводя его на экран и предоставляя возможность распечатать на чеке.
- «Снятие денег» – производит операцию снятия денег с карты. Для этого пользователь должен ввести сумму, которую он хочет снять. Клиент пошлет запрос на сервер о текущем балансеи получит ответ. Если на карте достаточно денег, то операция на сервере завершится успешно, и банкомат выдаст требуемую сумму. Также при нажатии на кнопку «Печать» будет напечатан чек по данной операции. Если обнаружится, что на карте недостаточно денег, то с карты ничего не снимется, и клиент выводит соответствующее сообщение на экран.

После возврата карты, пользователь может вставить ее опять либо уйти, нажав кнопку «Выход».

Также как в реальном банкомате операции общения с сервером обеспечиваются по сети. Подавтомат *AServer* удаленно вызывает необходимые методы сервера. При возникновении события «Ошибка при работе с сервером» клиент осуществляет возврат карты и переходит в начальное состояние.

Данный проект состоит из двух частей – клиентской и серверной. В клиентской части реализован пользовательский интерфейс(*AClient*), а также интерфейс отправки запросов на сервер(*AServer*). Серверная часть производит операции со счетами.

Роль сервера исполняет класс *Server*, написанный и запускающийся отдельно. Поведение клиента моделируется автоматом *AClient* и вложенным в него автоматом *AServer*.

На рис. 10 изображена схема связей автоматов *AClient* и *AServer*.

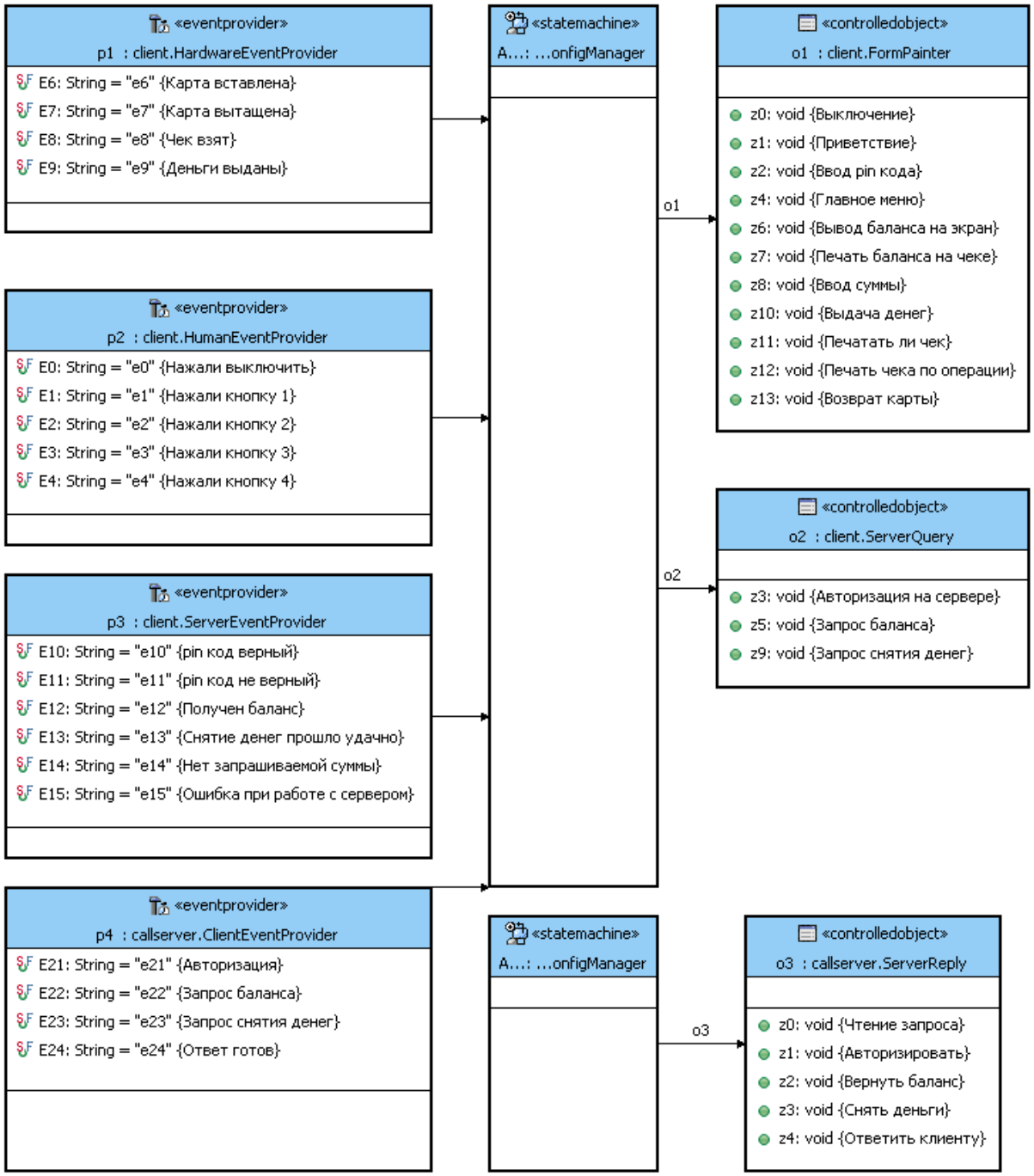

Рис. 10. Схема связей. Несмотря на отсутствие связи между автоматами, *AServer* вложен в *AClient*

На рис. 11 приведен граф переходов автомата *AClient*.

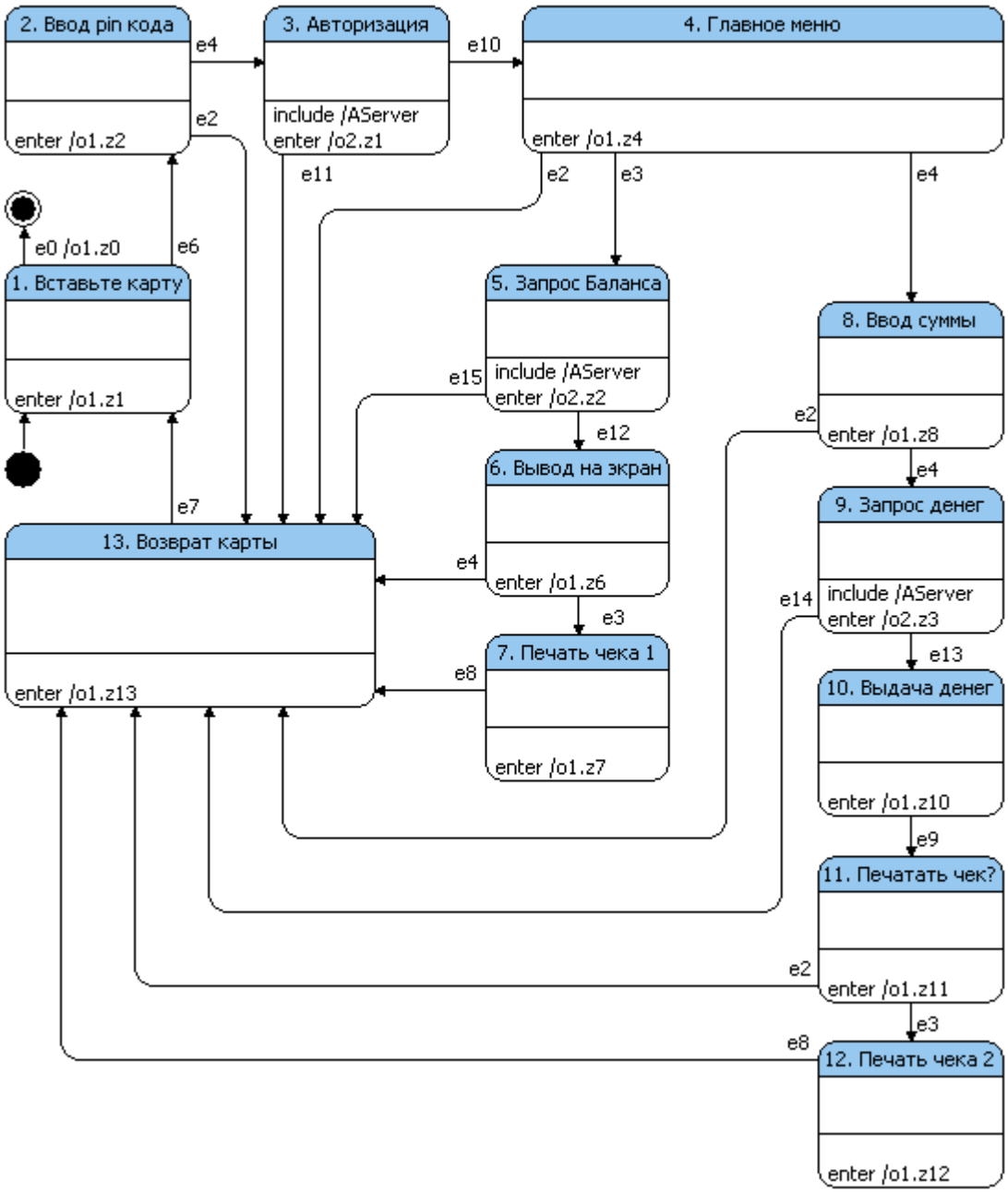

Рис. 11. Автомат `AClient`.

На рис. 12 приведен граф переходов автомата *AServer*, принимающего запросы от клиента.

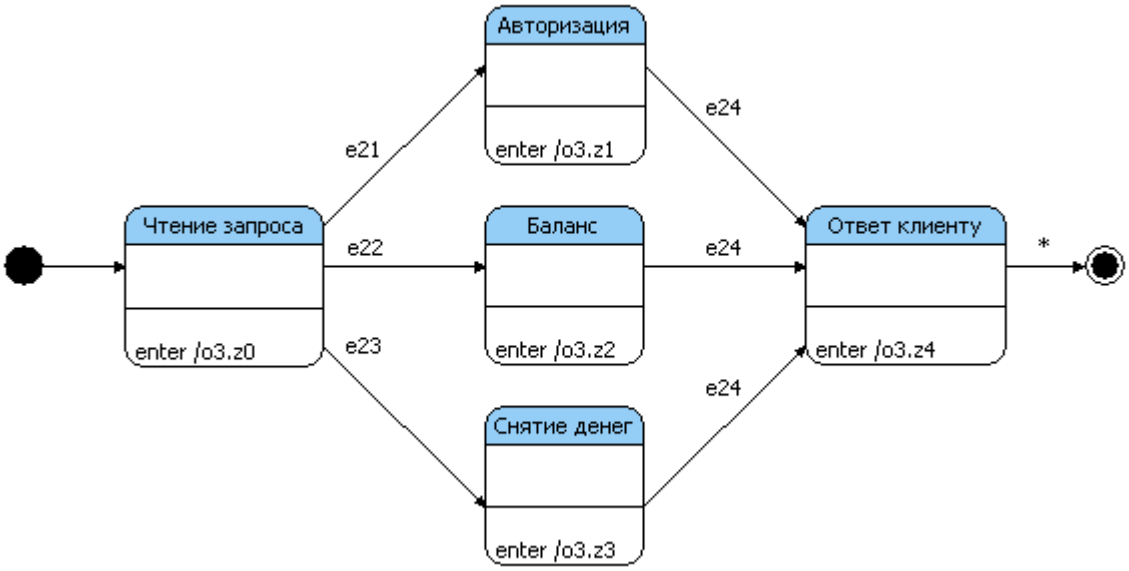

Рис. 12. Автомат *AServer*.

4.2. Верифицируемые свойства

Для демонстрации работы метода требуется сформулировать свойства, которые будут проверяться для приведенной визуальной автоматной программы. Для полноты проверки программы свойства должны быть двух типов:

- истинные свойства, которые позволяют выявить ошибки второго рода;
- ложные свойства, которые позволяют выявить ошибки первого рода. Кроме того, они позволяют продемонстрировать метод восстановления контрпримера из *модели Крипке* в автоматную модель.

Ложное свойство: «банкомат не выдает деньги».

Истинное свойство: «выдаваемая сумма не превышает остаток на счете».

4.3. Верификация

Для проведения верификации требуется знать, какие номера инструментальное средство *Converter* присвоило состояниям системы автоматов.

На рис. 13 показаны номера, которые *Converter* присвоил состояниям автомата *AClient*.

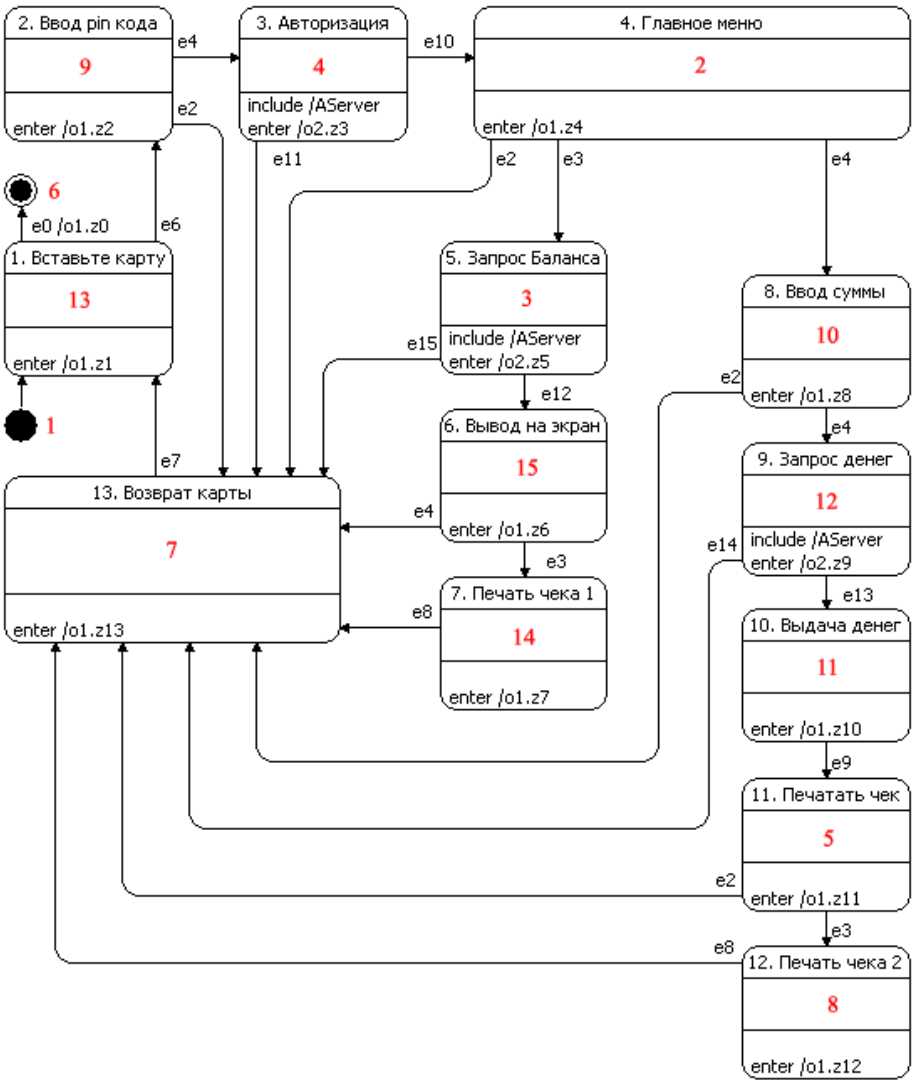

Рис. 13. Номера состояний автомата AClient

На рис. 14 показаны номера, которые *Converter* присвоил состояниям автомата *AServer*.

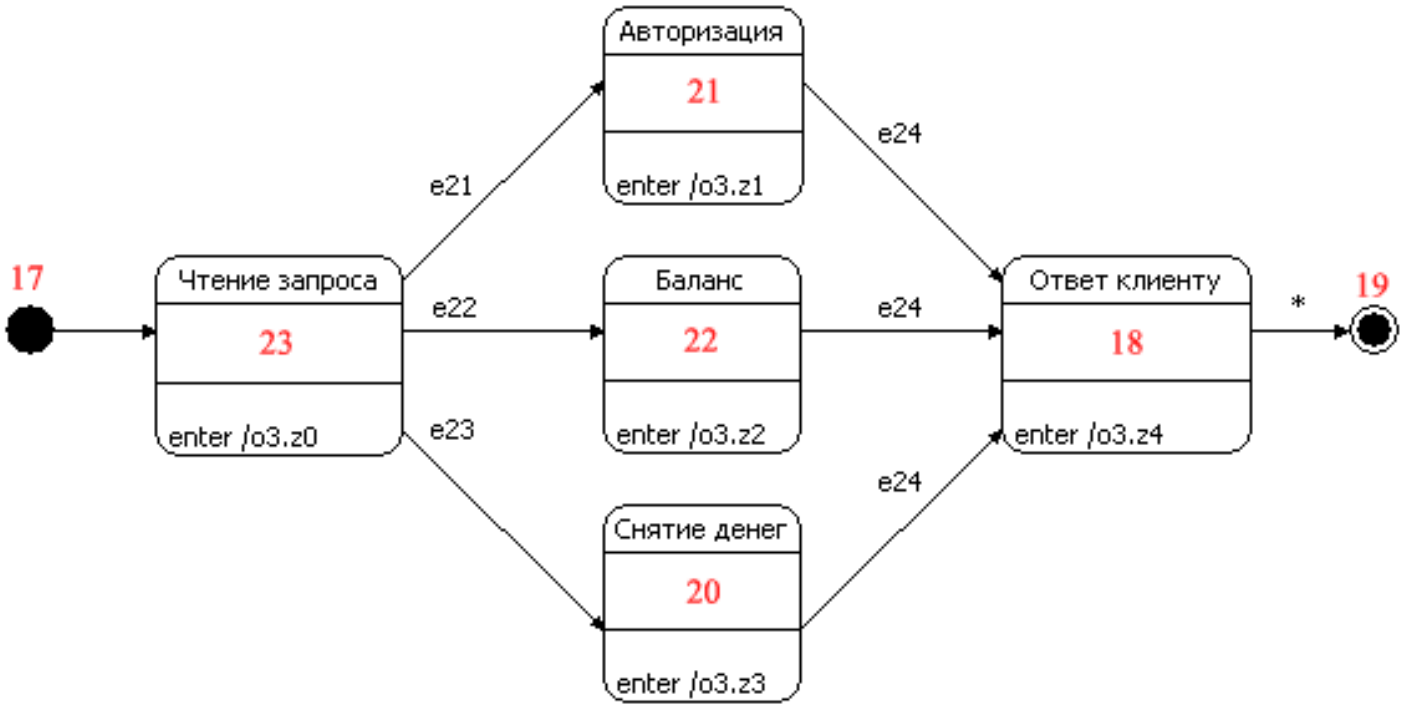

Рис. 14. номера состояний автомата AServer

4.3.1. Проверяемое свойство: «Банкомат не выдает деньги»

Напомним, что на вход верификатору *SPIN* требуется подавать отрицания проверяемых свойств. Запишем отрицание этого свойства на русском языке: «Банкомат когда-нибудь выдаст деньги». Выдача денег происходит в выходном воздействии *o1.z10*. Для построения формулы на языке *LTL*, соответствующей данному свойству, введем два атомарных высказывания:

`o == 1` и `z == 10`.

Первое высказывание обозначает, что используется объект управления с индексом 1, а второе обозначает, что было вызвано выходное воздействие с индексом 10. Перефразируем данное свойство (точнее, его отрицание) через введенные элементарные высказывания: «Автомат *AClient* когда-нибудь вызовет выходное воздействие № 10 объекта управления № 1». Следовательно, формула будет выглядеть следующим образом:

```
<>({o == 1} && {z == 10}).
```

Ожидаемый результат: в отчете о проведенной верификации должно быть сообщение об ошибке и выведен контрпример. В этом контрпримере должен быть путь по состояниям автомата *AClient* до момента, когда вызовется выходное воздействие *o1.z10*.

Проведем верификацию. Инструментальное средство *Converter* создаст следующий отчет:

```
Converter v. 2.0
verification with formula "<>({o == 1} && {z == 10})"
warning: for p.o. reduction to be valid the never claim
        must be stutter-invariant
(never claims generated from LTL formulae are stutter-
        invariant)
pan: claim violated! (at depth 2407)
pan: wrote bank2.trail
pan: reducing search depth to 1203
pan: end state in claim reached (at depth 2407)
pan: wrote bank2.trail
pan: reducing search depth to 1203
pan: claim violated! (at depth 959)
pan: wrote bank2.trail
pan: reducing search depth to 479
pan: end state in claim reached (at depth 959)
pan: wrote bank2.trail
pan: reducing search depth to 479
pan: claim violated! (at depth 337)
pan: wrote bank2.trail
pan: reducing search depth to 168
pan: end state in claim reached (at depth 337)
pan: wrote bank2.trail
pan: reducing search depth to 168
(Spin Version 4.2.8 -- 6 January 2007)
  + Partial Order Reduction

Full statespace search for:
  never claim                     +
  assertion violations            + (if within scope
        of claim)
  acceptance   cycles             + (fairness
        disabled)
  invalid end states              - (disabled by never
        claim)

State-vector 72 byte, depth reached 2407, errors: 6
    5930 states, stored
    9505 states, matched
   15435 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 38 (resolved)

5.809   memory usage (Mbyte)

spin: warning, "bank2" is newer than bank2.trail
Never claim moves to line 595     [(1)]
          State AClient 1 : s1
          Going to state AClient 13 : 1. Вставьте карту
          State AClient 13 : 1. Вставьте карту
          Action o1.z1
          Event = e6
```

```
            Going to state AClient 9 : 2. Ввод pin кода
            State AClient 9 : 2. Ввод pin кода
            Action o1.z2
            Event = e4
            Going to state AClient 4 : 3. Авторизация
            State AClient 4 : 3. Авторизация
            Action o2.z3
                State AServer 17 : s1
            Event = e10
            Going to state AClient 2 : 4. Главное меню
            State AClient 2 : 4. Главное меню
            Action o1.z4
            Event = e4
            Going to state AClient 10 : 8. Ввод суммы
            State AClient 10 : 8. Ввод суммы
            Action o1.z8
            Event = e4
            Going to state AClient 12 : 9. Запрос денег
            State AClient 12 : 9. Запрос денег
            Action o2.z9
            Event = e13
            Going to state AClient 11 : 10. Выдача денег
            State AClient 11 : 10. Выдача денег
            Action o1.z10
Never claim moves to line 594
          [(((o==1)&&(z==10)))]
Never claim moves to line 598      [(1)]
              State AServer 23 : Чтение запроса
              Action o3.z0
spin: trail ends after 338 steps
#processes: 3
    lastEvent = 13
    o = -1
    z = 10
    stateAClient = 11
    queue 3 (AClientAServer):
    stateAServer = 17
    queue 1 (epin):
    queue 2 (epout):
338:  proc  2 (eventProvider) line 561 "bank2" (state
        28)
338:  proc  1 (AServer) line 507 "bank2" (state 140)
338:  proc  0 (AClient) line 242 "bank2" (state 197)
338:  proc  - (:never:) line 599 "bank2" (state 8)
        <valid end state>
3 processes created
```

Строка отчета

```
State-vector 72 byte, depth reached 2407, errors: 6
```

говорит о том, что была найдена ошибка. Выпишем контрпример в явном виде:

Автомат *Aclient* перешел в состояние *s1*.
Автомат *Aclient* перешел в состояние *1. Вставьте карту*.
Было вызвано выходное воздействие *o1.z1*.
Произошло событие *e6*.
Автомат *Aclient* перешел в состояние *2. Ввод pin кода*.
Было вызвано выходное воздействие *o1.z2*.
Произошло событие *e4*.
Автомат *Aclient* перешел в состояние *3. Авторизация*.
Было вызвано выходное воздействие *o2.z3*.
Автомат *AServer* перешел в состояние *s1*.
Произошло событие *e10*.
Автомат *Aclient* перешел в состояние *4. Главное меню*.
Было вызвано выходное воздействие *o1.z4*.
Произошло событие *e4*.
Автомат *Aclient* перешел в состояние *8. Ввод суммы*.
Было вызвано выходное воздействие *o1.z8*.
Произошло событие *e4*.
Автомат *Aclient* перешел в состояние *9. Запрос денег*.
Было вызвано выходное воздействие *o2.z9*.
Произошло событие *e13*.
Автомат *Aclient* перешел в состояние *10. Выдача денег*.
Было вызвано выходное воздействие *o1.z10*.

Как и ожидалось, контрпример содержит путь по состояниям автомата *AClient* до состояния «10. Выдача денег».

4.3.2. Проверяемое свойство: «Выдаваемая сумма не превышает остаток на счете»

Проверяется отсутствие последовательности действий, при которой пользователь получает большую сумму денег, чем имеющаяся на счете.

Запишем отрицание этого свойства на русском языке: «Пользователь попытался снять больше денег, чем доступно по карте и произошла выдача денег». Когда пользователь пытается снять больше денег, чем доступно по карте, возникает событие *e14* («Нет запрашиваемой суммы»). Для *LTL*-формулы потребуются три элементарных высказывания:

`lastEvent == e14`, `o == 1` и `z == 10`.

LTL-формула для данного свойства выглядит следующим образом:

```
<>({lastEvent == e14} && ({lastEvent == e14} U ({o == 1} && {z
                          == 10}))).
```

Эта формула означает следующее: когда-нибудь произойдет событие *e14* и последнее произошедшее событие будет *e14* до тех пор, пока не произойдет выдача денег. В подформуле

```
({lastEvent == e14}) && ({lastEvent == e14} U ({o == 1} && {z == 10}))
```

требуется конструкция

```
({lastEvent == e14}) && ( … )
```

для того чтобы событие *e14* обязательно произошло, так как выражение *pUq* будет верно и в том случае, когда элементарное высказывание *p* в пути в *модели Крипке* не встретилось ни разу, а сразу встретилось *q*, что будет подтверждено ниже.

Ожидаемый результат: в отчете о проведенной верификации должно быть сказано, что ошибок нет.

Проведем верификацию. Инструментальное средство *Converter* создаст следующий отчет:

```
Converter v. 2.0
verification with formula "<>({lastEvent == e14} &&
        ({lastEvent == e14} U ({o == 1} && {z ==
        10})))"
warning: for p.o. reduction to be valid the never claim
        must be stutter-invariant
(never claims generated from LTL formulae are stutter-
        invariant)
(Spin Version 4.2.8 -- 6 January 2007)
  + Partial Order Reduction

Full statespace search for:
  never claim                     +
  assertion violations            + (if within scope
        of claim)
  acceptance   cycles             + (fairness
        disabled)
  invalid end states              - (disabled by never
        claim)

State-vector 72 byte, depth reached 5735, errors: 0
   28317 states, stored
   56105 states, matched
   84422 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 768 (resolved)

Stats on memory usage (in Megabytes):
2.265   equivalent memory usage for states
        (stored*(State-vector + overhead))
```

```
2.002   actual memory usage for states (compression:
        88.36%)
  State-vector as stored = 63 byte + 8 byte overhead
2.097   memory used for hash table (-w19)
3.200   memory used for DFS stack (-m100000)
0.056   memory lost to fragmentation
7.243   total actual memory usage

spin: cannot find trail file
```

Строка отчета

```
State-vector 72 byte, depth reached 5735, errors: 0
```

свидетельствует о том, что ошибок нет.

Проведем теперь верификацию банкомата с формулой

```
<>({lastEvent == e14} U {o == 1} && {z == 10}).
```

Ожидаемый результат: В отчете о проведенной верификации должны содержаться сообщение об ошибке и контрпример: трасса пути по автомату *AClient*, как показано на рис. 15.

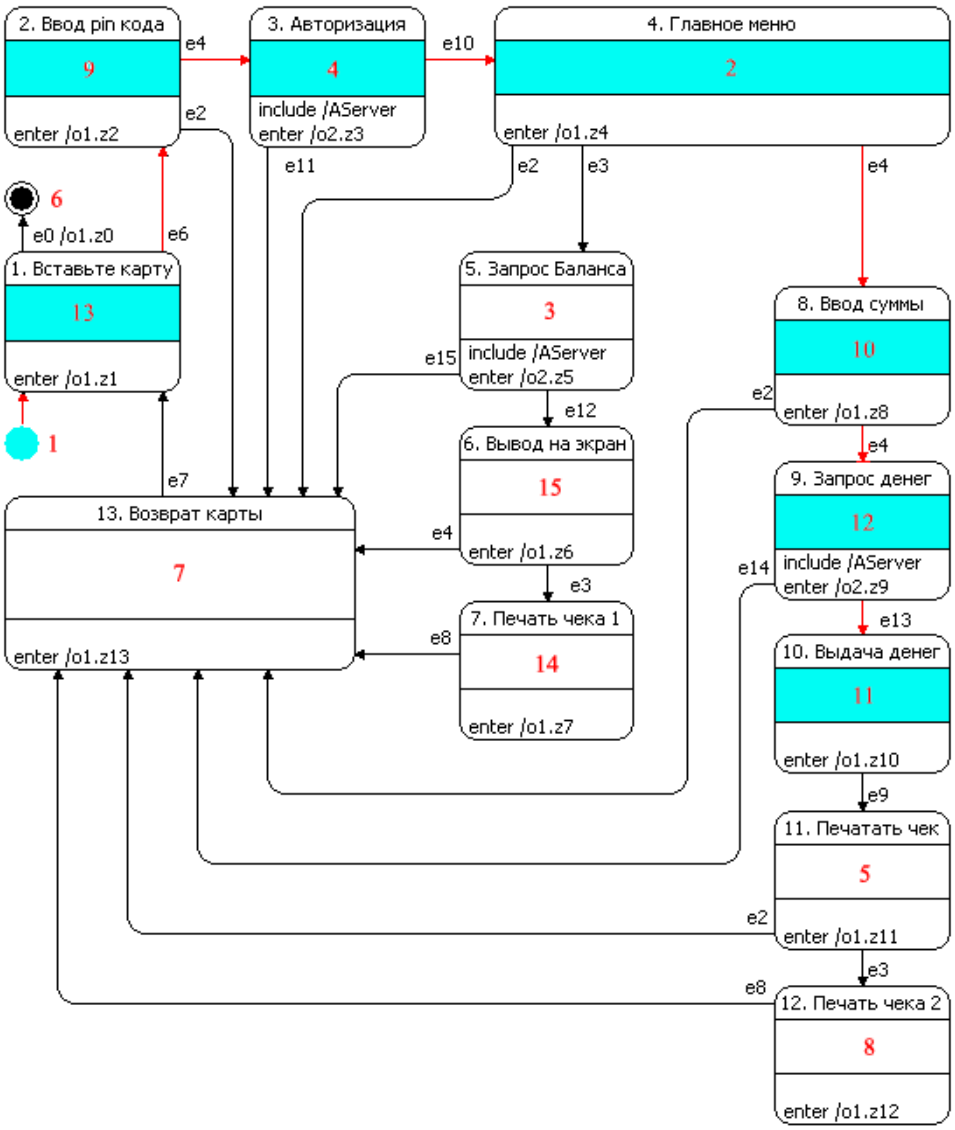

Рис. 15. Трасса ошибки к формуле <>({lastEvent == e14} U {stateAClient == 11})

Проведем верификацию. Инструментальное средство *Converter* выдаст следующий отчет:

```
Converter v. 2.0
verification with formula "<>({lastEvent == e14} U ({o
        == 1} && {z == 10}))"
warning: for p.o. reduction to be valid the never claim
        must be stutter-invariant
(never claims generated from LTL formulae are stutter-
        invariant)
pan: claim violated! (at depth 2407)
pan: wrote bank2.trail
pan: reducing search depth to 1203
pan: end state in claim reached (at depth 2407)
pan: wrote bank2.trail
pan: reducing search depth to 1203
pan: claim violated! (at depth 959)
pan: wrote bank2.trail
pan: reducing search depth to 479
```

```
pan: end state in claim reached (at depth 959)
pan: wrote bank2.trail
pan: reducing search depth to 479
pan: claim violated! (at depth 337)
pan: wrote bank2.trail
pan: reducing search depth to 168
pan: end state in claim reached (at depth 337)
pan: wrote bank2.trail
pan: reducing search depth to 168
(Spin Version 4.2.8 -- 6 January 2007)
  + Partial Order Reduction

Full statespace search for:
  never claim                   +
  assertion violations          + (if within scope
        of claim)
  acceptance   cycles           + (fairness
        disabled)
  invalid end states            - (disabled by never
        claim)

State-vector 72 byte, depth reached 2407, errors: 6
    5930 states, stored
    9505 states, matched
   15435 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 38 (resolved)

5.809   memory usage (Mbyte)

Never claim moves to line 596     [(1)]
          State AClient 1 : s1
          Going to state AClient 13 : 1. Вставьте карту
          State AClient 13 : 1. Вставьте карту
          Action o1.z1
          Event = e6
          Going to state AClient 9 : 2. Ввод pin кода
          State AClient 9 : 2. Ввод pin кода
          Action o1.z2
          Event = e4
          Going to state AClient 4 : 3. Авторизация
          State AClient 4 : 3. Авторизация
          Action o2.z3
              State AServer 17 : s1
          Event = e10
          Going to state AClient 2 : 4. Главное меню
          State AClient 2 : 4. Главное меню
          Action o1.z4
          Event = e4
          Going to state AClient 10 : 8. Ввод суммы
          State AClient 10 : 8. Ввод суммы
          Action o1.z8
          Event = e4
```

```
                Going to state AClient 12 : 9. Запрос денег
                State AClient 12 : 9. Запрос денег
                Action o2.z9
                Event = e13
                Going to state AClient 11 : 10. Выдача денег
                State AClient 11 : 10. Выдача денег
                Action o1.z10
        Never claim moves to line 595
                [(((o==1)&&(z==10)))]
        Never claim moves to line 599      [(1)]
                    State AServer 23 : Чтение запроса
                    Action o3.z0
        spin: trail ends after 338 steps
        #processes: 3
            lastEvent = 13
            o = -1
            z = 10
            stateAClient = 11
            queue 3 (AClientAServer):
            stateAServer = 17
            queue 1 (epin):
            queue 2 (epout):
        338:  proc  2 (eventProvider) line 562 "bank2" (state
                28)
        338:  proc  1 (AServer) line 508 "bank2" (state 140)
        338:  proc  0 (AClient) line 243 "bank2" (state 197)
        338:  proc  - (:never:) line 600 "bank2" (state 8)
                <valid end state>
        3 processes created
```

Строка отчета

```
State-vector 72 byte, depth reached 2407, errors: 6
```

свидетельствует о том, что была найдена ошибка. Таким образом, путь в автомате *AClient* соответствует состояниям и переходам, выделенным на рис. 15 голубым и красным цветами соответственно.

Выводы по главе 4

1. Проведена апробация разработанного в настоящей работе инструментального средства *Converter 2.0*.
2. Это средство выполняло верификацию всех проверенных свойств банкомата корректно.

Выводы по работе

1. В предложенном методе верификации автоматных программ используется верификатор *SPIN*. Этот верификатор является одним из наиболее мощных и известных. Его используют *NASA* [23] и многие другие организации, где требуется повышенная надежность.
2. Визуальные **автоматные программы** удобны для проектирования и наглядны. Кроме того, они позволяют **автоматически** построить *модель Крипке* и произвести верификацию.
3. Настоящая работа предлагает **метод автоматической верификации автоматных программ**, написанных в среде *UniMod* с помощью верификатора *SPIN*.
4. В настоящей работе был разработан универсальный метод верификации автоматных программ.
5. Разработанное в настоящей работе инструментальное средство *Converter 2.0* позволяет верифицировать автомат размером примерно в 20 раз больше, чем инструментальное средство *Converter 0.50*, разработанное автором в бакалаврской работе.

Источники

1. *Новиков Ф. А.* Визуальное конструирование программ // Информационно-управляющие системы. 2005. № 6, с. 9 – 22. http://is.ifmo.ru/works/visualcons/
2. *Шалыто А. А.* Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. – 628 с. http://is.ifmo.ru/books/switch/1/
3. *UniMod home page*: http://UniMod.sourceforge.net/
4. *Лифшиц Ю.* Верификация программ и темпоральные логики. Лекция № 3 курса «Современные задачи теоретической информатики». http://download.yandex.ru/class/lifshits/lecture-note03.pdf
5. *Лифшиц Ю.* Символьная верификация программ. Лекция № 4 курса «Современные задачи теоретической информатики». http://download.yandex.ru/class/lifshits/lecture-note04.pdf
6. *Кларк Э. М., Грамберг О., Пелед Д.* Верификация моделей программ: Model Checking. М.: МЦНМО, 2002. – 416 с.
7. *Holzman G. J.* Design And Validation Of Computer Protocols. Prentice Hall, 1991.
8. *Holzman G.J.* The model checker SPIN //IEEE Trans. on Software Engineering. Vol. 23, 1997. № 5, pp. 279 – 295.
9. *SPIN home page.* http://SPINroot.com
10. *Linear temporal logic.* http://en.wikipedia.org/wiki/Linear_temporal_logic
11. *Васильева К. А., Кузьмин Е. В.* Верификация автоматных программ с использованием LTL //Моделирование и анализ информационных систем. 2007. № 1. с. 3 – 14. http://is.ifmo.ru/verification/_LTL_for_SPIN.pdf
12. *Büchi automaton.* http://en.wikipedia.org/wiki/Büchi_automaton
13. *Васильева К. А., Кузьмин Е. В., Соколов В. А.* Верификация автоматных программ с использованием логики LTL. http://is.ifmo.ru/verification/_ltl_aut_ver_1.pdf
14. *Dijkstra E.W.* Guarded commands, non-determinacy and formal derivation of programs // CACM. 18. 1975. № 8.
15. *Vardy M. Y., Wolper P.* An automata-theoretic approach to automatic program verification /Proc. 1-st IEEE Symp. On Logic in Computer Science. 1986.
16. *Promela reference – ltl.* http://www.spinroot.com/spin/Man/ltl.html
17. *Описание Промелы и некоторых примеров программ* http://sevik.ru/mstu/docs/promela-rtf.zip
18. *Лукин М. А., Шалыто А. А.* Верификация автоматных программ использованием верификатора SPIN // Научно-технический вестник СПбГУ ИТМО. Вып. 53. Автоматное программирование. 2009, с.145 – 161. http://books.ifmo.ru/ntv/ntv/53/ntv_53.pdf
19. *Гуров В. С., Яминов Б. Р.* Верификация автоматных программ при помощи верификатора *UniMod.Verifier* // Научно-технический вестник СПбГУ ИТМО. Вып. 53. Автоматное программирование. 2009, с. 162 – 176. http://books.ifmo.ru/ntv/ntv/53/ntv_53.pdf

20. *Егоров К. В., Шалыто А. А.* Разработка верификатора автоматных программ //Научно-технический вестник СПбГУ ИТМО. Вып. 53. Автоматное программирование. 2009, с. 177 – 188. http://books.ifmo.ru/ntv/ntv/53/ntv_53.pdf
21. *Яминов Б. Р.* Сравнение методов верификации *UniMod*-моделей. Магистерская диссертация, СПбГУ ИТМО. 2009.
22. *Отчет по контракту о верификации автоматных программ. Второй этап.* http://is.ifmo.ru/verification/_2007_02_report-verification.pdf
23. *Риган П., Хемилтон С.* NASA: миссия надежна. http://www.osp.ru/os/2004/03/184060/