Санкт-Петербургский государственный университет информационных технологий, механики и оптики

М. А. Лукин

Верификация автоматных программ

Бакалаврская работа

Научный руководитель – профессор А. А. Шалыто

Санкт-Петербург
2007

Оглавление

Введение

Настоящая работа посвящена созданию методики автоматической верификации автоматных программ [1]. Практическим ее приложением явился инструмент, позволяющий производить верификацию спроектированных и реализованных на инструментальном средстве *UniMod* [2] автоматных программ. В данной работе верификация производится с помощью метода *Model Checking* [3 – 7].

Верификация модели программы — это один из основных методов поиска ошибок в программе. Для того чтобы произвести верификацию программы, требуется:

1. Построить формальную модель, приемлемую для инструментальных средств верификации моделей. Обычно при построении модели абстрагируются от несущественных деталей программы для 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* [6–10]. Важным вопросом является *полнота* спецификации.
3. Произвести 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 наиболее популярным верификатором является *SPIN* [6 – 8]. Его популярность позволяет надеяться на то, что он не содержит ошибок. На вход этого верификатора подаются модель программы, описанная на языке *Promela* [8], и требования к модели, записанные на языке *LTL*. Верификатор по модели программы строит модель *Крипке* [3, 4], а по инверсии каждого требования – автомат Бюхи [11]. 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

Цель работы — разработать методику для автоматической верификации автоматных программ, написанных в среде *UniMod*. Для этого необходимо решить следующие задачи:

1. Разработать методику записи автоматной модели на языке *Promela*.
2. Разработать инструмент автоматической трансляции автоматной модели, разработанной в среде *UniMod*, в модель, описанную на языке *Promela*.
3. Обеспечить преобразование контрпримера, построенного верификатором *SPIN*, в автоматную модель.

Глава 2. Основные понятия

2.1. Синтаксис языка *Promela*

Promela — это специализированный язык высокого уровня, который используется верификатором *SPIN*. Его синтаксис напоминает синтаксис языка *C*. Модель на языке *Promela* состоит из следующих элементов:

- объявления типов данных;
- объявления каналов передачи переменных;
- объявления переменных;
- объявления процессов;
- процесса *init.*

Процесс отдаленно можно рассматривать как процедуру, выполняемую в отдельном потоке. Пример 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 у него стоит 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

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 в том случае, если `x` меньше семи. В противном случае он останавливает выполнение процесса до тех пор, пока значение `x` не станет меньше семи. Некоторые операторы, например, оператор присваивания, *выполнимы* всегда.

Язык *Promela* содержит также операторы ветвления и цикла, синтаксис которых основан на охраняемых командах Дейкстры [12]. Их синтаксис приведен в табл. 1.

Таблица 1. Синтаксис операторов ветвления и цикла

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

Его следует понимать так: если выполнено условие $guard_i$, то выполнить действие S_i. В случае, если выполняются несколько условий, то происходит недетерминированный выбор одного из них. Если все условия не выполняются, то выполняется действие S_k, соответствующее *else*. Конструкция *else* может отсутствовать. При этом если не справедливо ни одно условие, то выполнение процесса блокируется до тех пор, пока хотя бы одно из них не начнет выполняться.

2.2. Модель *Крипке*

Пусть *AP* — множество атомарных высказываний. Модель *Крипке* [3,4] над *AP* — это четверка $M = (S, S_0, R, L)$, в которой:

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

Модель *Крипке* приспособлена для верификации. Для записи требований к ней используются языки темпоральной логики: *LTL*, *CTL*, *CTL** и другие.

2.3. Язык *LTL*

Язык *LTL* (*Linear-Time Logic*) состоит из множества атомарных высказываний $p_1, p_2, \ldots \in AP$, логических связей ¬, /\, \/, →, и темпоральных операторов. Пусть φ — правильно построенная формула. Тогда

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

тоже правильно построенные формулы.

Операторы `G` и `F` необходимы для упрощения формул. Их можно выразить через оператор `U`:

- `F`$\varphi \equiv$ `1U`φ;
- `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 верификатор *SPIN* не поддерживает оператор `X neXt)`[1], так как этот верификатор генерирует вспомогательные и служебные состояния в модели *Крипке*.

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

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

2.4. 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 [11, 13, 14]. Более того, он может строиться автоматически.

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

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

```
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. 1). Двойная линия обозначает допустимое состояние.

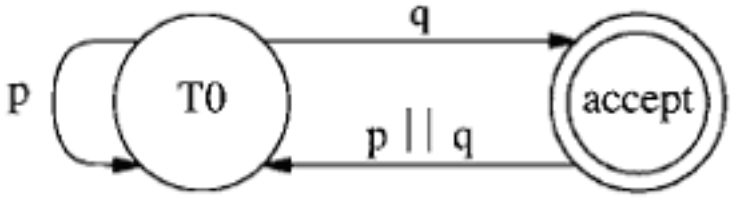

Рис. 1. Автомат Бюхи для формулы *G(p U q)*

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*.

Глава 3. Результаты, полученные в работе

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. Все это подать на вход верификатору *SPIN*.
4. Проанализировать отчет, выданный верификатором *SPIN*, и построить контрпример.

3.1.2. Построение модели на языке *Promela*

В данном разделе приводится алгоритм построения модели на языке *Promela* по автоматной программе.

1. Подготовка

Для каждого автомата A_i вводится переменная $stateA_i$. На языке *Promela* это записывается так:

```
int stateAi;
```

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

```
stateAi = k;
```

Событие *exx* (*xx* — номер события) на переходе между состояниями описывается в модели следующим образом:

```
lastEvent = xx;
```

Для каждого автомата A_i создадим функцию. На языке *Promela* это записывается так:

```
inline Ai() {
/* тело функции */
}
```

2. Построение

1. Присвоить каждому состоянию автомата A_i свой номер.
2. Найти начальное состояние s. Присвоить $stateA_i = s$.
3. Построить цикл:

   ```
   do
   ::(stateAi == s1) ->
   ::(stateAi == s2) ->
   …
   ::(stateAi == sk) ->
   od;
   ```

   где s_1, …, s_k — номера состояний автомата A_i.
4. Если в некоторое состояние s_j вложен автомат Am, то в условие $(stateA_i == s_j)$ дописывается вызов функции этого автомата:

   ```
   Am();
   ```
5. Для каждого состояния s_j найти все возможные переходы (s_j, s_l) из него. К условию $(stateA_i == s_j)$ дописать конструкцию *if*. Для каждого перехода (s_j, s_l) в конструкции *if* дописывается следующее:

   ```
   ::stateAi = sl;
   ```

   Это означает, что для присваивания $stateA_i == s_l$ необходимое условие выполнено всегда. Если этот переход помечен событием *exx*, то выражение

   ```
   lastEvent = exx;
   ```

   дописывается после предыдущего выражения.

   Таким образом, получается конструкция вида:

   ```
   if
   …
   ```

```
::stateAi = sl;
  lastEvent = exx;
...
fi;
```

6. Если состояние s_t — конечное, то в условие *(stateA*$_i$ == s_t*)* дописывается инструкция завершения цикла:

```
break;
```

Построение модели рассмотрим на примере графа переходов автомата на рис. 2, который входит в систему из трех автоматов. Этот автомат имеет три состояния. В виду того, что рассматриваемый автомат третий, а предыдущие два содержат шесть состояний, то *стартовому* состоянию в рассматриваемом автомате присвоим номер семь, а *конечному* – восемь. При этом *главному интерфейсному состоянию* присвоим номер девять.

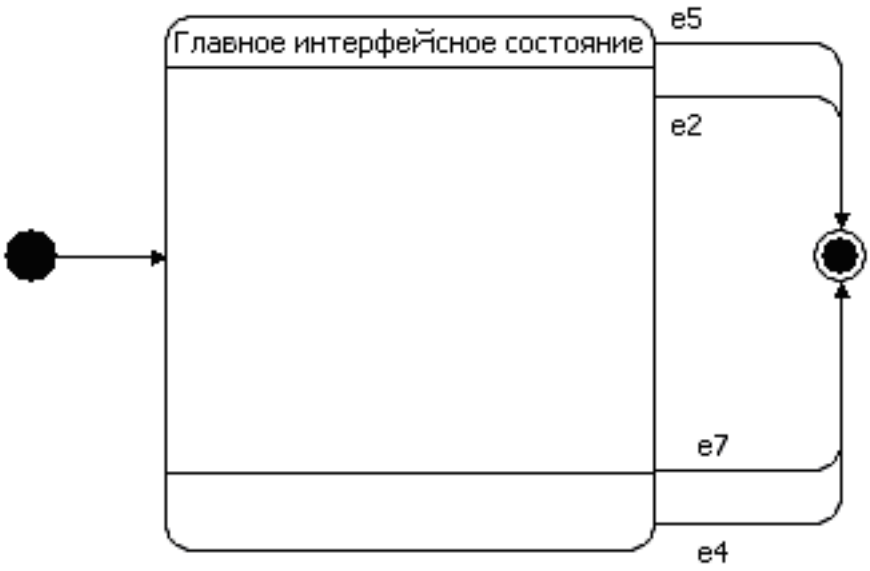

Рис. 2. Автомат *A3*

По графу переходов сгенерируем на основе изложенной методики модель на языке *Promela*:

```
int stateA3;
inline A3() {
```

```
    stateA3 = 7;
    do
    ::(stateA3 == 7) ->
        if
            ::stateA3 = 9;
        fi;
    ::(stateA3 == 8) ->
        break;
    ::(stateA3 == 9) ->
        if
            ::stateA3 = 8;
              lastEvent = 4;
            ::stateA3 = 8;
              lastEvent = 7;
             ::stateA3 = 8;
              lastEvent = 2;
             :stateA3 = 8;
              lastEvent = 5;

        fi;
    od;
}
```

3.1.3. Преобразование *LTL*-формул

Каждому элементарному высказыванию присвоим идентификатор p_k. В модель запишем следующий код:

```
#define pk <элементарное_высказывание>
```

После этого заменим все элементарные высказывания их идентификаторами. В таком виде *LTL*-формула может быть обработана верификатором *SPIN*, при условии, что темпоральные операторы записаны в нотации *SPIN*.

3.1.4. Построение контрпримера

Верификатор *SPIN* автоматически строит контрпример как путь в модели, поданной ему на вход. Так как модель на языке *Promela* эквивалентна исходному автомату, обратное преобразование не требуется.

3.2. Формализация типовых требований к автоматным моделям

Рассмотрим типовые требования, которые можно применять ко всем автоматным моделям [15].

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, но и 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 методики

3.3.1. Описание инструментальное средство *Converter*, разработанного в настоящей работе

Это инструментальное средство является практической реализацией предложенной методики верификации автоматных программ и позволяет производить **полностью автоматическую верификацию** автоматных программ, не считая того, что контрпример выводится в текстовый файл, а не отображается в *UniMod*.

По автоматной программе *Converter* автоматически создает модель на языке *Promela*, а *LTL*-формула автоматически преобразовывается к виду, пригодному для верификатора *SPIN*.

Инструментальное средство должно получать на вход три параметра: путь к *XML*-описанию автоматной программы, имя файла, в который записывается созданная модель и одну *LTL*-формулу с требованиями к модели (ее спецификацией).

На выходе инструментального средства *Converter* выдается файл *report.txt*, в котором записан полный отчет о проведенной верификации, включая автоматически построенный верификатором *SPIN* контрпример, представленный в текстовом виде.

3.3.2. Описание работы инструментального средства

Автоматическая верификация автоматных программ состоит из нескольких этапов (рис. 3).

1. Сначала *Converter* принимает на вход *XML*-файл с описанием автоматной программы, созданный в среде *UniMod*, и требования к ней, записанные на языке *LTL* (рис. 4). **Важно: верификатору на вход подаются не требования, а их отрицание.**

2. При помощи *UniMod* из *XML* получается автоматная модель на языке *Java*.
3. *Converter* транслирует автоматную модель с языка *Java* на язык *Promela*.
4. *Converter* преобразует *LTL*-формулу к виду, понятному верификатору *SPIN* (рис. 3) по следующей схеме:
    - 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. Высказывание

      ```
      {lastEvent == e13},
      ```

      преобразовываются в следующую строку на *Promela*:

      ```
      #define pk (lastEvent == 13),
      ```

      где *k* — номер элементарного высказывания.
5. *Converter* запускает *SPIN* в режиме генерации конструкции *never claim* (с помощью команды `SPIN -f <формула>`). *SPIN* по *LTL*-формуле генерирует конструкцию n*ever claim*, представляющую собой автомат Бюхи, записанный на языке *Promela*.

6. После того, как на языке *Promela* описаны модель и требование к ней, *Converter* запускает *SPIN* на верификацию (командой `SPIN -a <Модель>`). *SPIN* генерирует файл *pan.c*, представляющий собой программу на языке *C*.
7. После компиляции получается верификатор для данной конкретной модели с заданной спецификацией. При обнаружении ошибок эта программа выдает *trail*-файл, в котором описана трасса ошибки в формате, понятном *SPIN*.

По команде `SPIN -t -p <Модель>` *SPIN* выводит контрпример. *Converter* помещает отчет, созданный *SPIN* в файл `report.txt`.

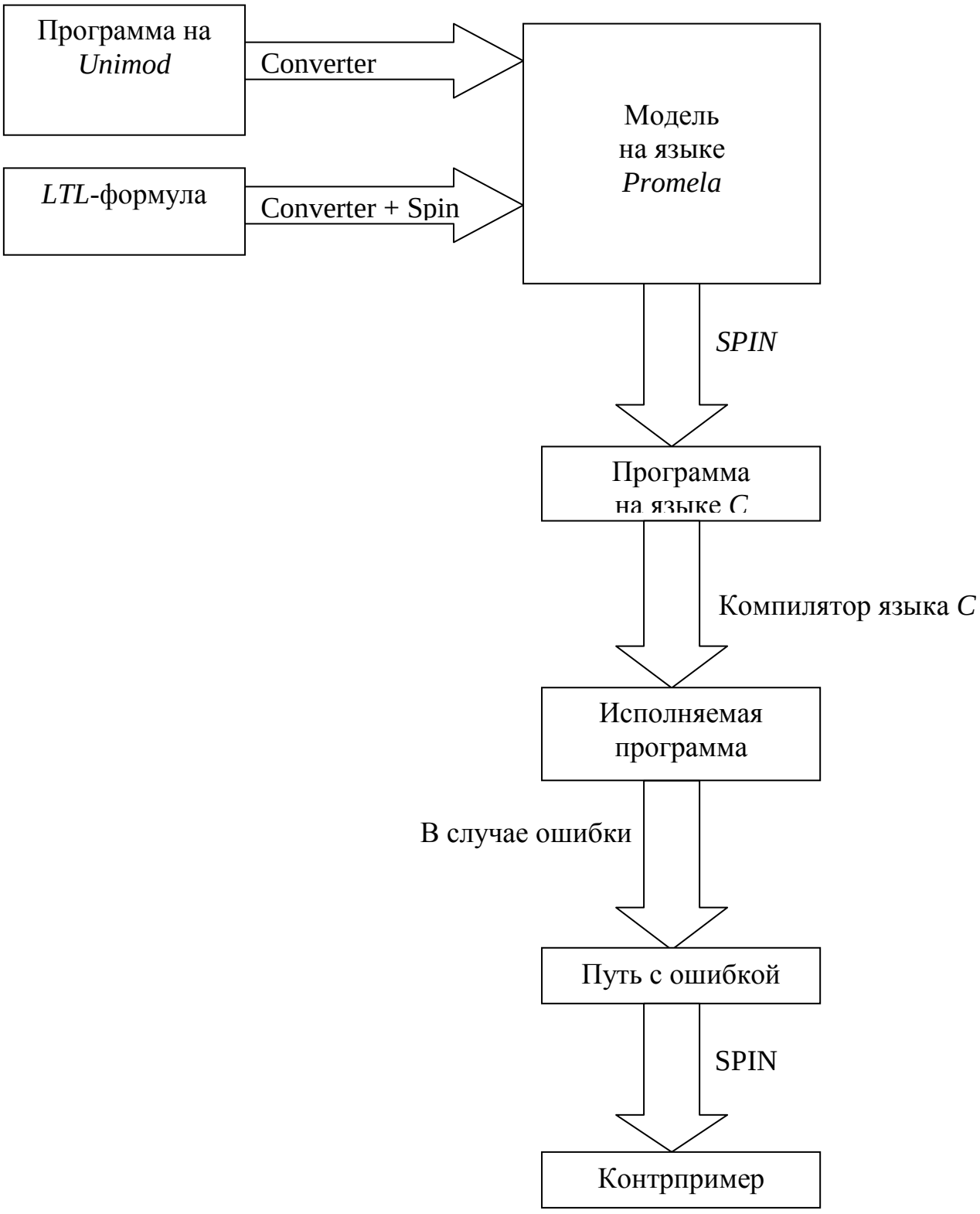

Рис. 3. Описание работы инструментального средства

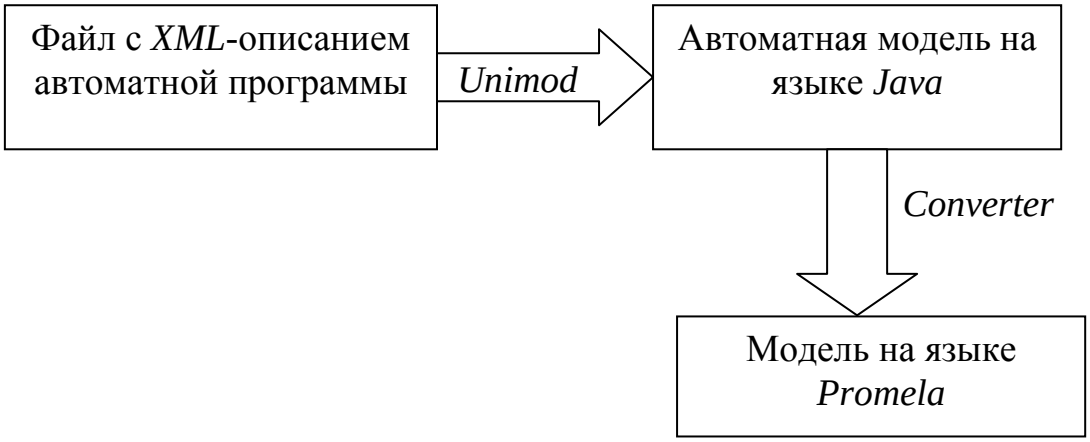

Рис. 4. Преобразование автоматной программы

3.3.3. Обратное преобразование контрпримера

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

Строка отчета

```
StateAk = s,
```

где *stateAk* — состояние автомата *Ak*, *s* — номер состояния, обозначает вход автомата *Ak* в состояние *s*.

Строка отчета

```
lastEvent = exx,
```

где *exx* — некоторое событие обозначает, что был совершен переход по событию *exx*.

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

3.5. Сравнение с существующими решениями

В настоящее время разработаны несколько методов верификации автоматных программ [10,16,17]. Однако все они не очень эффективны, так как требуют ручного задания модели программы. Предложенный в данной работе метод избавлен от этого недостатка.

Глава 4. Пример

4.1. Постановка задачи

Рассмотрим автоматную реализацию игры *Ним* [18] (рис. 5). *Ним* — игра для двух игроков, каждый из которых по очереди делает ход. Перед игроками располагается поле с фишками. Известны различные варианты этой игры. В данном проекте правила игры таковы:

- фишки раскладываются в несколько рядов;
- игроки по очереди забирают фишки из любого ряда;
- не разрешается за один ход брать фишки из нескольких рядов;
- за один ход игрок должен взять хотя бы одну фишку;
- выигрывает тот, кто возьмет последнюю фишку (фишки).

В данном примере была использована одна из первых версий реализации с одним автоматом. Цифрами указаны номера состояний, присвоенные инструментом *Converter*.

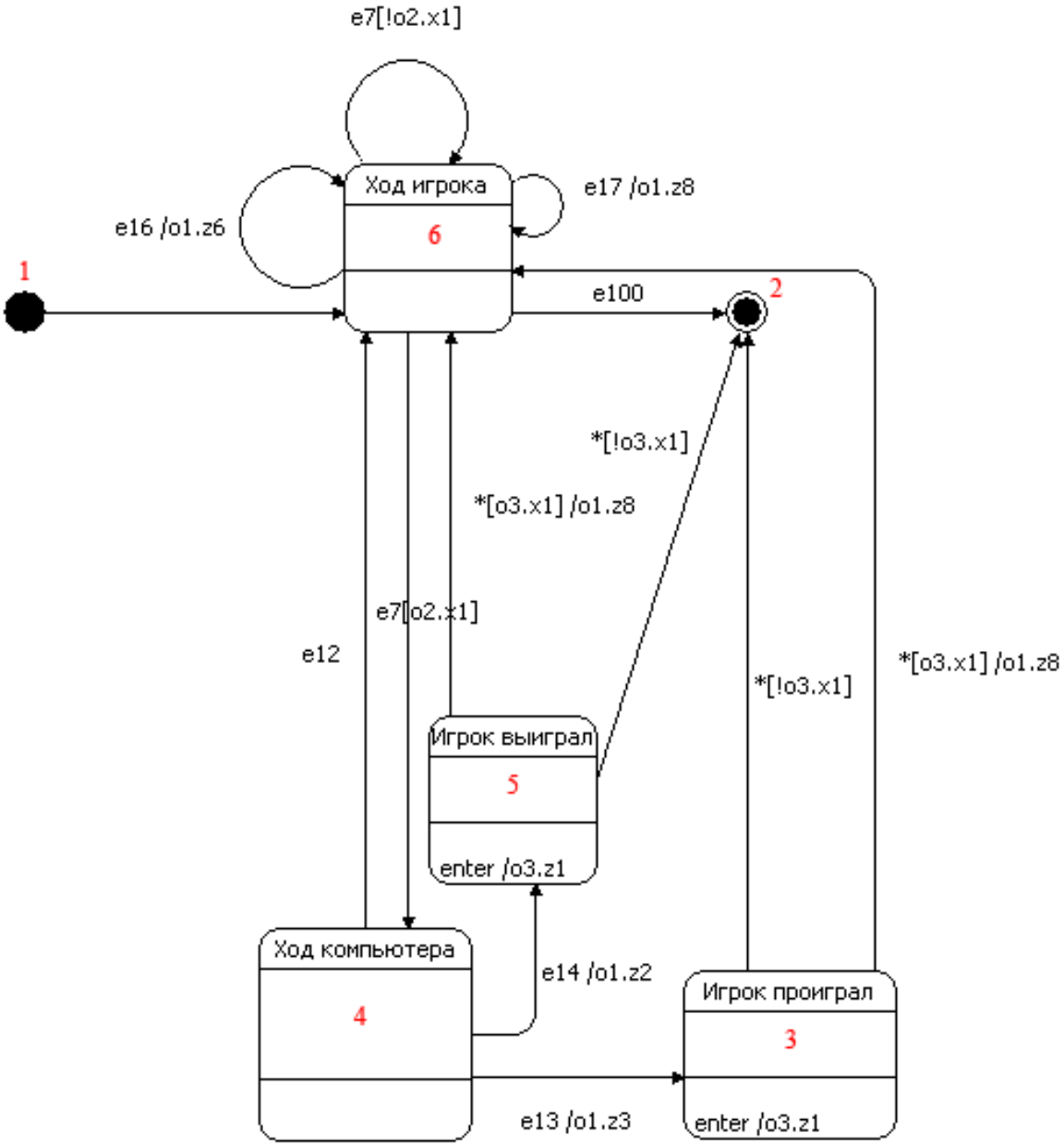

Рис. 5. Автоматная реализация игры *Ним*

Допустим, в ней была допущена ошибка (рис. 6). Лишний переход выделен жирным.

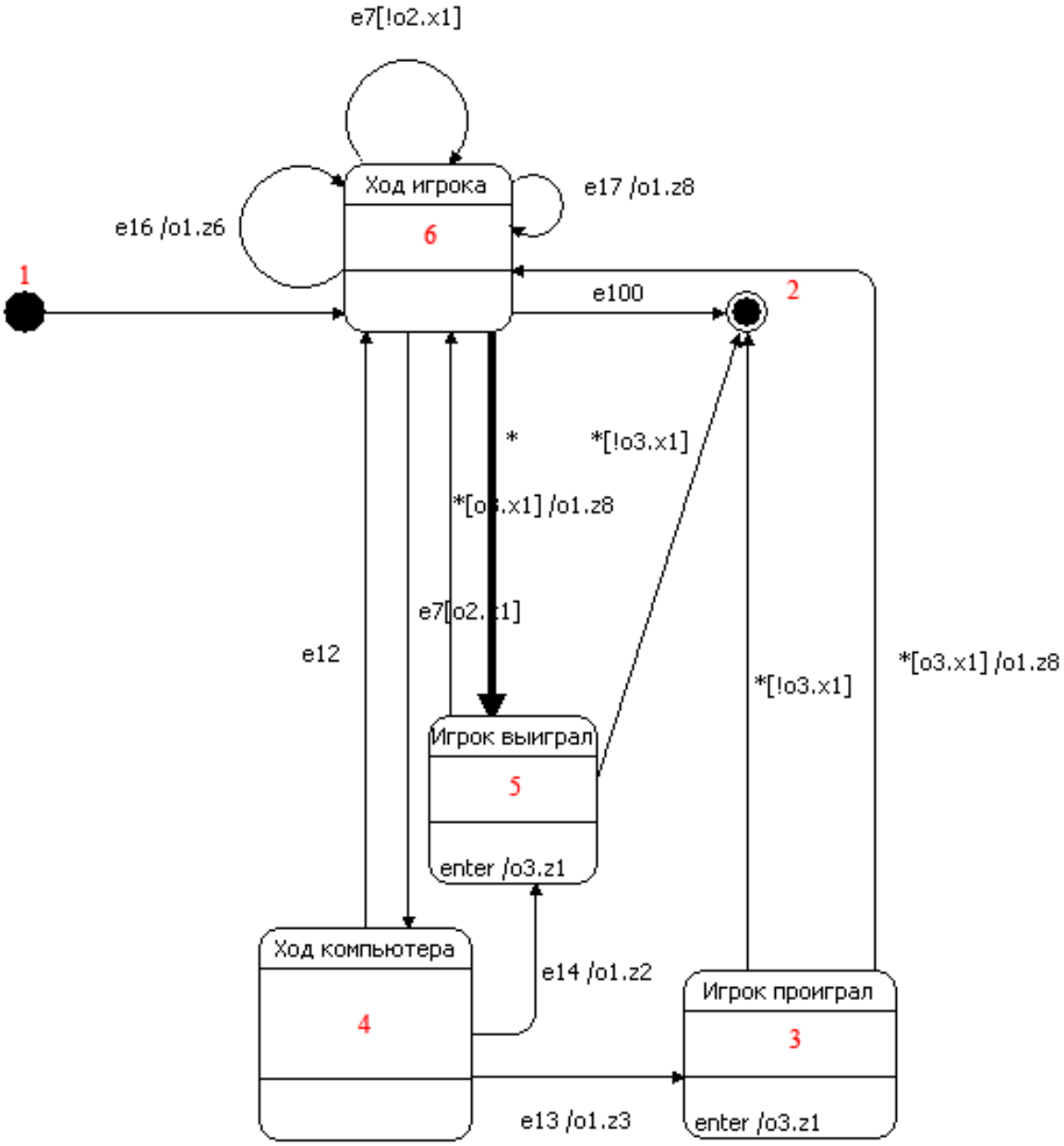

Рис. 6. Реализация с ошибкой

4.2. Верификация

В данной реализации решение о том, кто выиграл, принималось в состоянии *Ход компьютера*. Запишем это требование на языке *LTL*:

$$\neg (stateA1 \; != \; 4) \; U \; (stateA1 \; == \; 5).$$

Запишем его в формате, подходящем *Converter:*

```
!({stateA1 != 4} U {stateA1 == 5}).
```

Подадим на вход *Converter* неправильный автомат и отрицание сформулированного требования. *Converter* по автомату построит следующую модель:

```
#define p1    (stateA1 != 4)
#define p2    (stateA1 == 5)

int lastEvent;
#define STATE_0   0 /*Top*/
#define STATE_1   1 /*s2*/
#define STATE_2   2 /*s3*/
#define STATE_3   3 /*Игрок проиграл*/
#define STATE_4   4 /*Ход компьютера*/
#define STATE_5   5 /*Игрок выиграл*/
#define STATE_6   6 /*Ход игрока*/

int stateA1;
inline A1() {
    stateA1 = STATE_1;
    do
    ::(stateA1 == STATE_1) ->
        printf("State 1 : s2\n");
        if
            ::stateA1 = STATE_6;
        fi;
    ::(stateA1 == STATE_2) ->
        printf("State 2 : s3\n");
        break;
    ::(stateA1 == STATE_3) ->
        printf("State 3 : Игрок проиграл\n");
        if
            ::stateA1 = STATE_2;
            ::stateA1 = STATE_6;
        fi;
    ::(stateA1 == STATE_4) ->
        printf("State 4 : Ход компьютера\n");
        if
            ::stateA1 = STATE_3;
              lastEvent = 13;
            ::stateA1 = STATE_6;
              lastEvent = 12;
            ::stateA1 = STATE_5;
              lastEvent = 14;
        fi;
    ::(stateA1 == STATE_5) ->
```

```
        printf("State 5 : Игрок выиграл\n");
        if
            ::stateA1 = STATE_6;
            ::stateA1 = STATE_2;
        fi;
    ::(stateA1 == STATE_6) ->
        printf("State 6 : Ход игрока\n");
        if
            ::stateA1 = STATE_6;
              lastEvent = 16;
            ::stateA1 = STATE_4;
              lastEvent = 7;
            ::stateA1 = STATE_6;
              lastEvent = 7;
            ::stateA1 = STATE_6;
              lastEvent = 17;
            ::stateA1 = STATE_2;
              lastEvent = 100;
            ::stateA1 = STATE_5;
        fi;
    od;
}
proctype Model() {
    A1();
}

init {
run Model();
}
never {    /* p1 U p2 */
T0_init:
    if
    :: ((p2)) -> goto accept_all
    :: ((p1)) -> goto T0_init
    fi;
accept_all:
    skip
}
```

В приведенном тексте жирным шрифтом выделен неправильный переход.

Верификатор *SPIN* по этой модели построит программу на языке *C*. Результатом работы этой программы является сообщение о числе ошибок и, если ошибки найдены, *trail*-файл. В данном случае *trail*-файл выглядит так:

-2:2:-2

-4:-4:-4

1:0:55

2:1:51

3:0:55

4:2:0

5:0:55

6:2:1

7:0:55

8:2:2

9:0:55

10:2:3

11:0:55

12:2:31

13:0:55

14:2:32

15:0:55

16:2:43

17:0:53

18:2:25

19:0:59

20:1:52

Верификатор *SPIN* обрабатывает этот файл и строит контрпример, а разработанное инструментальное средство *Converter* все собирает воедино и строит отчет:

```
pan: claim violated! (at depth 19)
pan: wrote models/incorr.ltl.trail
```

```
(SPIN Version 4.2.8 -- 6 January 2007)
Warning: Search not completed
    + Partial Order Reduction

Full statespace search for:
    never claim          +
    assertion violations + (if within scope of claim)
    acceptance   cycles  - (not selected)
    invalid end states   - (disabled by never claim)

State-vector 24 byte, depth reached 27, errors: 1
      20 states, stored
       4 states, matched
      24 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

2.622    memory usage (Mbyte)
```

Начало контрпримера.

```
Starting :init: with pid 0
Starting :never: with pid 1
Never claim moves to line 75  [((stateA1!=4))]
```

Отображение перехода автомата *Бюхи*.

```
Starting Model with pid 2
  2:proc  0 (:init:) line  69 "models/incorr.ltl"
(state 1)    [(run Model())]
  4:proc  1 (Model) line  15 "models/incorr.ltl"
(state 1)    [stateA1 = 1]
```

Автомат *A1* переходит в состояние 1 (начальное состояние).

```
  6:proc  1 (Model) line  17 "models/incorr.ltl"
(state 2)    [((stateA1==1))]
              State 1 : s2
  8:proc  1 (Model) line  18 "models/incorr.ltl"
(state 3)    [printf('State 1 : s2\\n')]
 10:proc  1 (Model) line  20 "models/incorr.ltl"
(state 4)    [stateA1 = 6]
```

Автомат *A1* переходит в состояние 6 (Ход игрока).

```
 12:proc  1 (Model) line  47 "models/incorr.ltl"
(state 2)    [((stateA1==6))]
              State 6 : Ход игрока
 14:proc  1 (Model) line  48 "models/incorr.ltl"
(state 33)   [printf('State 6 : Ход игрока\\n')]
 16:proc  1 (Model) line  60 "models/incorr.ltl"
(state 44)   [stateA1 = 5]
```

Автомат *A1* переходит в состояние 5 (Игрок выиграл).

```
Never claim moves to line 74  [((stateA1==5))]
 18:proc  1 (Model) line  41 "models/incorr.ltl"
(state 26)   [((stateA1==5))]
Never claim moves to line 78  [(1)]
SPIN: trail ends after 20 steps
#processes: 2
        lastEvent = 0
        stateA1 = 5
 20:proc  1 (Model) line  42 "models/incorr.ltl"
(state 27)
 20:proc  0 (:init:) line  70 "models/incorr.ltl"
(state 2) <valid end state>
 20:proc  - (:never:) line  79 "models/incorr.ltl"
(state 8) <valid end state>
```

```
2 processes created
```

Для удобства в отчете жирным шрифтом выделены моменты входа в новое состояние, а также сообщение о числе ошибок. С основным текстом идут комментарии к отчету.

В отчете программы *Converter* появился текст «errors: 1», сообщающий об ошибке.

4.3. Анализ контрпримера

Проанализируем отчет, выданный инструментом *Converter*. В начале каждой строки трассы содержится номер процесса и его имя в круглых скобках (Например, `proc 1 (Model)`). После этого записан номер строки в модели на языке *Promela* и имя файла с моделью (`line 15 "models/incorr.ltl"`). Далее в круглых скобках написан номер состояния в модели *Крипке*, построенной *SPIN* (`(state 1)`). В квадратных скобках указан код на языке *Promela* (`[stateA1 = 1]`).

Таким образом, из отчета следует, что контрпримером является путь (рис. 7): начальное состояние — состояние 6 (Ход игрока) — состояние 5 (Игрок выиграл).

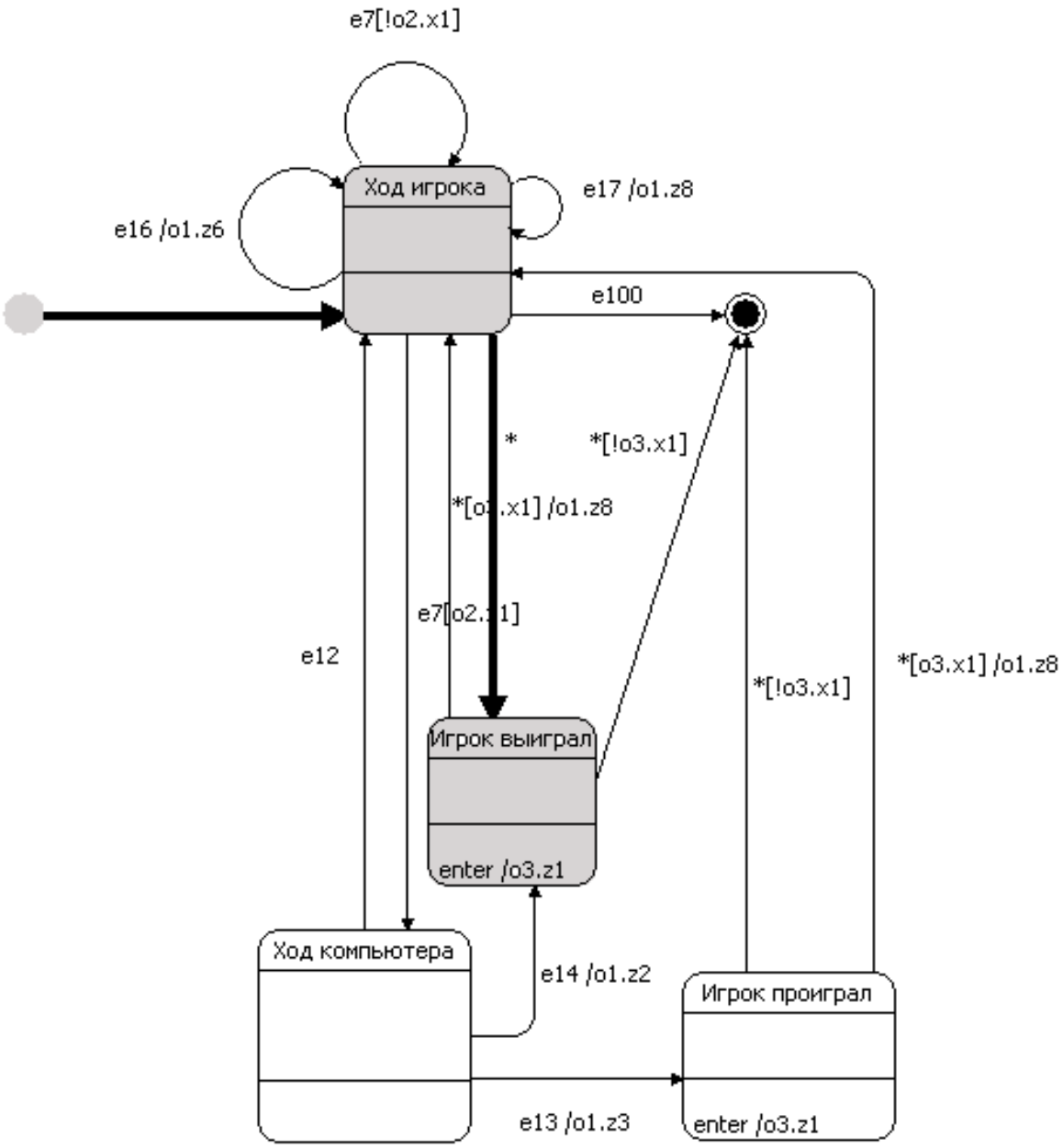

Рис. 7. Контрпример

Подадим теперь на вход *Converter* правильный автомат и такое же требование. Строка из отчета «errors: 0» говорит о том, что ошибка была исправлена.

Заключение

В предложенном методе верификации автоматных программ используется верификатор *SPIN*. Этот верификатор — один из наиболее мощных и известных верификаторов. Его используют *NASA* [18] и многие другие организации, где требуется повышенная надежность.

Автоматные программы удобны для проектирования и наглядны. Кроме того, они позволяют автоматически построить модель *Крипке* и произвести верификацию.

Настоящая работа предлагает метод автоматической верификации автоматных программ, написанных в среде *UniMod* с помощью верификатора *SPIN*.

Источники

1. *Шалыто А. А.* Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. – 628 с. – http://is.ifmo.ru/books/switch/1/
2. *UniMod home page*: http://UniMod.sourceforge.net/
3. *Лифшиц Ю.* Верификация программ и темпоральные логики. Лекция №3 курса «Современные задачи теоретической информатики». http://download.yandex.ru/class/lifshits/lecture-note03.pdf
4. *Лифшиц Ю.* Символьная верификация программ. Лекция № 4 курса «Современные задачи теоретической информатики». http://download.yandex.ru/class/lifshits/lecture-note04.pdf
5. *Кларк Э. М., Грамберг О., Пелед Д.* Верификация моделей программ: Model Checking. М.: МЦНМО. 2002. – 416 с.
6. *Holzman G.J.* Design And Validation Of Computer Protocols. NJ: Prentice Hall, 1991.
7. *Holzman G.J.* The model checker SPIN //IEEE Trans. on Software Engineering. 1997. № 4, pp. 279–295.
8. *SPIN home page*. http://SPINroot.com
9. *Linear temporal logic*. http://en.wikipedia.org/wiki/Linear_temporal_logic
10. *Васильева К. А., Кузьмин Е. В.* Верификация автоматных программ с использованием LTL. http://is.ifmo.ru/verification/_LTL_for_SPIN.pdf
11. *Büchi automaton*. http://en.wikipedia.org/wiki/Büchi_automaton
12. *Dijkstra, E.W.* Guarded commands, non-determinacy and formal derivation of programs //CACM. 1975. 8.
13. *Кюзара В. Е.* Реализация системы проверки моделей раскрашенных сетей Петри с использованием разверток. www.iis.nsk.su/preprints/pdf/094.pdf
14. *Vardy M.Y., Wolper P.* An automata-theoretic approach to automatic program verification / Proc. 1st IEEE Symp. On Logic in Computer Science. 1986.

15. *Описание Промелы и некоторых примеров программ* http://sevik.ru/mstu/docs/promela-rtf.zip
16. *Вельдер С. Э.* Введение в верификацию автоматных программ на основе метода *Model checking*. http://is.ifmo.ru/verification/modelchecking/
17. *Виноградов Р. А., Кузьмин Е. В., Соколов В. А.* Верификация автоматных программ средствами CPN/Tools. http://is.ifmo.ru/verification/_ver_prog.pdf
18. *Яковлев А. В., Лукин М. А., Шалыто А. А.* Реализация классической игры «Ним» на основе автоматного подхода. http://is.ifmo.ru/UniMod-projects/knim/
19. *NASA: миссия надежна.* http://www.osp.ru/os/2004/03/184060/