Вариант статьи опубликован в научно-техническом сборнике «Системы управления и обработки информации». 2010. Вып. 21, с. 84 – 94.

УДК 004.4’242

А.О. Ремизов, А.А. Шалыто, д-р техн. наук, профессор

ВЕРИФИКАЦИЯ АВТОМАТНЫХ ПРОГРАММ НА ОСНОВЕ МЕТОДА *MODEL CHECKING*

Проблема верификации систем промышленного и военного назначения возникла с момента их появления. Такие системы являются объектами повышенной ответственности, и поэтому к ним предъявляются высокие требования по качеству.

Большинство задач, решаемых сегодня в таких системах, реализуются с помощью программного обеспечения (ПО). Поэтому задача верификации ПО является одной из важнейших.

Верификацией называется процесс проверки соответствия создаваемых в ходе разработки и сопровождения ПО документов, моделей и других информационных сущностей, другим, ранее созданным или используемым в качестве исходных данных, а также соответствие этих сущностей и процессов их разработки правилам и стандартам [1].

Из этого определения следует, что верифицировать можно не только программный код, но также различные документы и модели на соответствие друг другу. Таким образом, верификация является надежным способом контроля качества процесса разработки ПО.

В настоящей работе верификация выполняется применительно к моделям, разработанным с использованием парадигмы автоматного программирования [2]. Этот подход к проектированию программного обеспечения основан на расширенной модели конечных автоматов и ориентирован на создание широкого класса приложений. Метод позволяет описывать поведение программ с помощью автоматов. Задача начальной загрузки информационно-управляющей системы, реализованной с помощью автоматного подхода и используемая в настоящей работе в качестве примера для верификации, рассмотрена в работе [3].

В настоящей работе перечислены методы верификации программного обеспечения, и показано, что для автоматных программ уровень автоматизации процесса верификации существенно выше, чем для программ, написанных традиционным способом, так как при автоматном подходе первичной является высокоуровневая модель поведения программы, а при верификации традиционных программ модель при помощи рефакторинга строится по уже готовой программе, что весьма трудоемко. Ненамного лучше обстоят дела, если создание программы начинается с написания модели программы на языке верификатора.

МЕТОДЫ ВЕРИФИКАЦИИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В работе [1] выполнен обзор методов верификации ПО, которые классифицируются следующим образом.

- Экспертиза. Методы верификации, в которых оценка артефактов жизненного цикла ПО выполняется людьми, непосредственно анализирующими эти артефакты.
- Статический анализ (например, проверка компилятором кода на наличие ошибок). Используется для проверки выполнения формализованных правил корректности построения артефактов жизненного цикла ПО и поиска типичных ошибок в них на основе некоторых шаблонов.
- Динамические методы (например, тестирование). Используют результаты работы реальной системы (или ее прототипа) для проверки соответствия этих результатов требованиям и проектным решениям.
- Формальные методы (например, на основе математических доказательств). Используют формальные модели требований, поведения и окружения ПО для анализа его свойств.
- Синтетические методы. Сочетают техники нескольких вышеприведенных типов.

ВЕРИФИКАЦИЯ АВТОМАТНЫХ ПРОГРАММ

Для верификации программ, выполненных с применением автоматного подхода, наиболее целесообразно использовать метод *Model Checking* [4]. Этот метод состоит в том, что на формальной модели программы проверяется выполнение формул темпоральной логики, каждая из которых соответствует одному из функциональных требований спецификации программы. Для автоматных программ при использовании метода *Model Checking* уровень автоматизации может быть существенно повышен.

При этом для автоматных программ преобразование описания поведения программы в модель для верификации может выполняться **автоматически**. Таким образом, исключается возможность ошибки или неправильного трактования поведения проверяемой программы, характерная для преобразования программы в модель, написанной традиционным способом. Для автоматных программ также отсутствует необходимость вручную создавать модель на языке верификатора. Кроме того, автоматные программы обладают тем преимуществом, что от контрпримера в программу также можно вернуться **автоматически**.

При создании программ рассматриваемого класса состояния делятся на **управляющие и вычислительные**, причем число состояний первого типа порядка нескольких сотен, а второго – значительно большая величина. При верификации автоматных программ

проверяются управляющие состояния, что делает задачу обозримой, а решение понятным человеку.

Диаграммы состояний автоматов являются частью проектной документации. Поэтому уже на стадии проектирования имеется возможность верифицировать разрабатываемое ПО и тем самым повысить его качество.

ВЕРИФИКАЦИЯ АВТОМАТНЫХ ПРОГРАММ С ПОМОЩЬЮ ИНСТРУМЕНТАЛЬНОГО СРЕДСТВА *SPIN*

В работах [5 – 8] описаны методики верификации автоматных программ методом *Model Checking*, который состоит из четырех этапов.

1. **Построение формальной модели.** При построении модели по готовой программе на основе рефакторинга трудоемкость верификации весьма высока. Если готовая программа отсутствует, то ее создание начинается с написания вручную модели, используя язык верификатора, что, по мнению авторов, неестественно. При проектировании автоматных программ их поведение задается диаграммами состояний, что позволяет автоматически и изоморфно переходить к модели на языке верификатора. После верификации диаграммы состояний могут автоматически преобразовываться в код.
2. **Формулировка требований к программе и модели с помощью формул темпоральной логики.** Для автоматных программ переход от требований к программе к требованиям к модели достаточно прост, что нельзя сказать о программах других классов.
3. **Верификация модели.** По сравнению с традиционными программами для автоматных программ автоматизирован переход от контрпримеров (трасс ошибок) к описанию автоматов.
4. **Анализ контрпримера при его наличии.** Проводится корректировка автоматной модели и повторная верификация.

Одним из популярных инструментальных средств верификации программ является свободно распространяемый верификатор *SPIN* [9]. Этапы верификации автоматной модели с его помощью состоят в следующем.

1. Автоматизированное преобразование автомата в текстовое описание модели (структуру Крипке) на языке *Promela*.
2. Задание **отрицания** проверяемого требования формулой темпоральной логики и генерации для нее конструкции *never claim*, которая представляет собой автомат Бюхи, записанный на языке *Promela*, и эквивалентный

проверяемой темпоральной формуле автомат. Для этого необходимо выполнить команду `spin -f <формула>`.

3. Преобразование текстового описания модели на языке *Promela* с включенной конструкцией *never claim* в программу на языке *C*. Для этого необходимо выполнить команду `spin -a <модель>`. При этом будет сгенерирован файл `pan.c`, соответствующей указанной программе.
4. После компиляции и запуска этой программы производится верификация модели. Строится пересечение автомата *never claim* и структуры Крипке. Пересечение строится «на лету», без ожидания полного построения структуры Крипке [8]. Программа `pan` выдает отчет о верификации и, если пересечение не пусто, создается *trail*-файл в формате верификатора *SPIN*, содержащий контрпример.
5. Если была найдена ошибка и, следовательно, создан *trail*-файл, то *SPIN* обрабатывает его и с помощью команды `spin -t <модель>` выводит отчет, содержащий контрпример.

ПРЕОБРАЗОВАНИЕ ДИАГРАММЫ СОСТОЯНИЙ АВТОМАТА В МОДЕЛЬ ВЕРИФИКАТОРА *SPIN*

Автоматизированное преобразование диаграммы состояний в текстовое описание модели на языке *Promela* выполняется в два этапа.

1. Диаграмма состояний проектируется с помощью инструментального средства *SwitchDesigner*, разработанного авторами, и сохраняется в *XML*-файл формата *SwitchDesigner*.
2. На основе разработанного *XSL*-шаблона осуществляется преобразование *XML*-файла диаграммы состояний в эквивалентное описание на языке *Promela* (для верификации и применения языков *C* и *C++* для написания исходного кода). В настоящей работе для этих преобразований использовалась утилита *XMLStarlet* [10].

Таким образом, по диаграмме состояний генерируется текст описания модели на языке *Promela* и исходный код на языках *C* и *C++*.

ВЕРИФИКАЦИЯ АВТОМАТА НАЧАЛЬНОЙ ЗАГРУЗКИ

На рис. 1 в качестве примера приведена схема связей автомата начальной загрузки. Эта схема позволяет использовать в диаграммах состояний только символы переменных, а не идентификаторы. Применение этого типа диаграмм, отсутствующих в *UML*, делает диаграммы состояний компактными и обозримыми [3]. Для объектно-ориентированных программ диаграммы классов могут представляться в виде схем связей, как это сделано в инструментальном средстве *UniMod*.

Диаграмма состояний автомата начальной загрузки изображена на рис. 2. Диаграмма отражает логику поведения программы. В дальнейшем по диаграмме изоморфно генерируется программный код на априори заданном языке программирования. Поэтому результаты верификации модели являются справедливыми для автоматной программы, сгенерированной по этой модели.

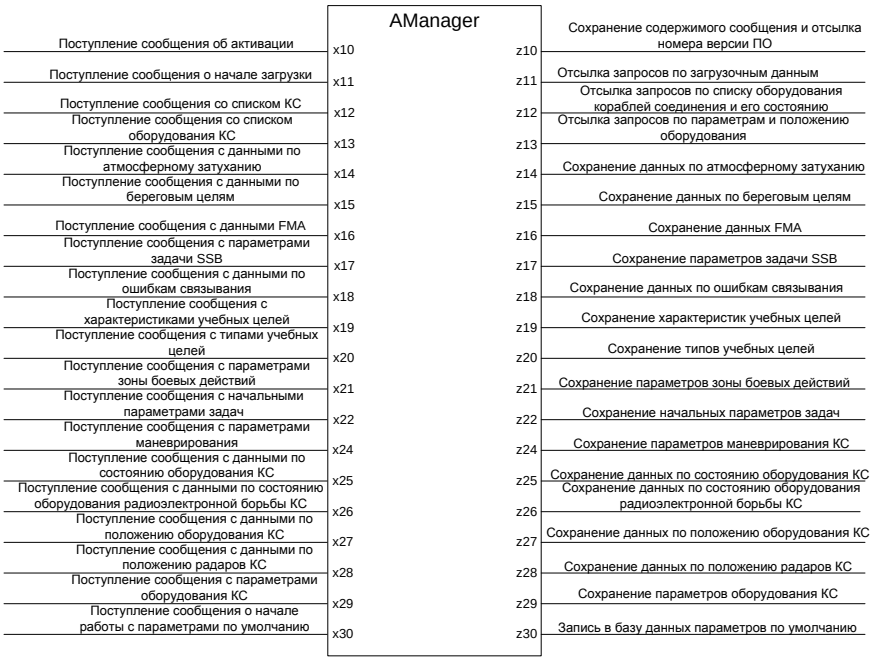

Рис. 1. Диаграмма связей автомата задачи начальной загрузки

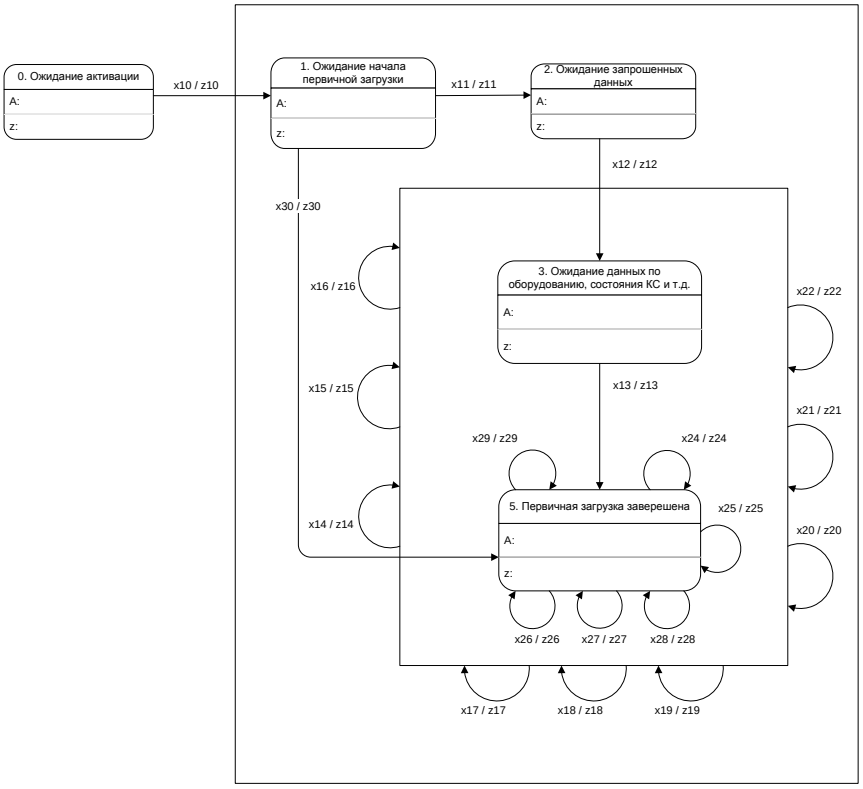

Рис. 2. Диаграмма состояний автомата задачи начальной загрузки

В таблице представлен перечень требований в автоматных терминах, и для каждого из них записана формула темпоральной логики, соответствующая отрицанию требования.

Таблица

Требования к программе для задачи начальной загрузки

№	Требование к программе	Требование к модели	Отрицание требования к модели	Формула темпоральной логики для отрицания требования к модели
1	Модуль активируется только при поступлении сообщения об активации	Входное воздействие `x10` активно до тех пор, пока автомат не перейдет в состояние 1	Любое другое воздействие, кроме `x10` активно до тех пор, пока автомат не перейдет в состояние 1	`<>(({{lastEvent > 10}) && (({lastEvent > 10}) U ({stateA0 == 1})))`

2	Загрузка в любом случае будет завершена	Состояние 5 автомата когда-нибудь будет достигнуто	Состояние 5 автомата никогда не будет достигнуто	`!(<>({stateA0 == 5}))`
3	Во время ожидания загрузки данных по оборудованию, модуль должен реагировать на получение данных FMA	В состоянии 3 автомат реагирует на входное воздействие `x16`	Никогда не будет получено входного воздействия `x16` в состоянии 3	`!(<>(({stateA0 == 3}) && ({lastEvent == 16})))`

Рассмотрим пример выполнения первого требования из таблицы.

1. По диаграмме состояний автомата, приведенной на рис. 2, генерируется текст описания модели на языке *Promela*, который представлен в листинге 1.

Листинг 1. Фрагмент текста описания модели на языке Promela для автомата задачи начальной загрузки

```
inline A0() {
  stateA0=0;
  do
  ::(stateA0==0)->
    printf("State 0: Waiting for activation\n");
    if
      ::stateA0=1;
      lastEvent=10;
      printf("Going to state 1\n");
    fi
  ::(stateA0==1)->
    printf("State 1: Waiting for persistent data loading start\n");
    if
      ::stateA0=2;
      lastEvent=11;
      printf("Going to state 2\n");

      ::stateA0=5;
      lastEvent=30;
      printf("Going to state 5\n");
    fi
  ...
  od
}

proctype Model()
{
```

```
  A0();
}

init {
  run Model();
}
```

2. Формула темпоральной логики, соответствующая отрицанию рассматриваемого требования, преобразуется в конструкцию *never claim* (листинг 2).

Листинг 2. Конструкция *never claim* для темпоральной формулы

```
<>(({lastEvent>10}) && (({lastEvent>10}) U ({stateA0 == 1})))
```

```
#define p1 (lastEvent>10)
#define p2 (stateA0 == 1)

never {    /* <>(p1 && (p1 U p2)) */
T0_init:
        if
        :: ((p1) && (p2)) -> goto accept_all
        :: ((p1)) -> goto T0_S4
        :: (1) -> goto T0_init
        fi;
T0_S4:
        if
        :: ((p2)) -> goto accept_all
        :: ((p1)) -> goto T0_S4
        fi;
accept_all:
        skip
}
```

3. По полученным описаниям генерируется файл *pan.c*.
4. После компиляции файла *pan.c* и запуска программы выводится отчет (листинг 3).

Листинг 3. Отчет программы *pan* (ошибок не обнаружено)

```
warning: never claim + accept labels requires -a flag to fully verify
hint: this search is more efficient if pan.c is compiled -DSAFETY
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 5.2.2 -- 7 September 2009)
        + 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 28 byte, depth reached 85, errors: 0
      118 states, stored
      110 states, matched
      228 transitions (= stored+matched)
        0 atomic steps
hash conflicts: 0 (resolved)

    2.501 memory usage (Mbyte)
```

В приведенном примере отсутствуют ошибки, о чем говорит строка отчета

```
State-vector 28 byte, depth reached 85, errors: 0.
```

Поскольку верификация завершилась без обнаружения ошибок, программа *pan* не создает *trail*-файла.

Внесем в диаграмму состояний, изображенную на рис. 2, ошибку – заменим входное воздействие `x10` на `x11` (рис. 3).

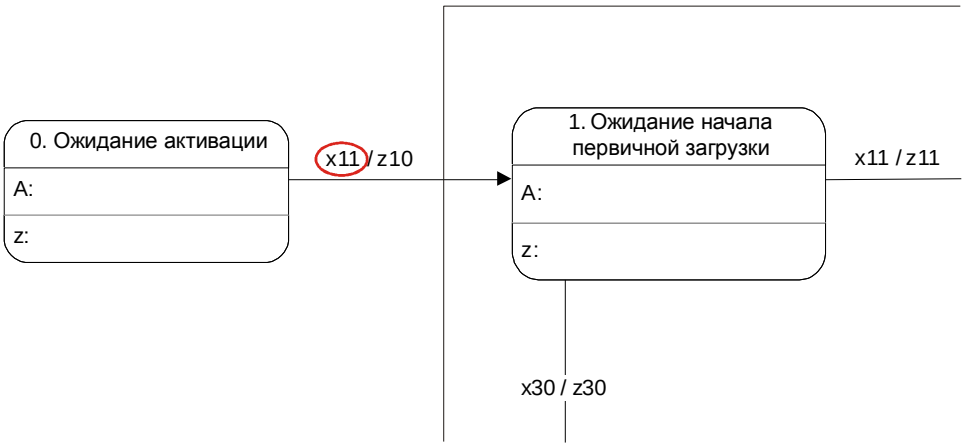

Рис. 3. Фрагмент диаграммы с внесенной ошибкой

Проводя проверку рассматриваемого требования повторно, получим отчет, приведенный в листинге 4.

Листинг 4. Отчет программы *pan* (обнаружена ошибка)

```
warning: never claim + accept labels requires -a flag to fully verify
hint: this search is more efficient if pan.c is compiled -DSAFETY
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 15)
pan: wrote SWC1.trail

(Spin Version 5.2.2 -- 7 September 2009)
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 28 byte, depth reached 15, errors: 1
        8 states, stored
```

```
        0 states, matched
        8 transitions (= stored+matched)
        0 atomic steps
hash conflicts: 0 (resolved)

    2.501 memory usage (Mbyte)
```

Строка отчета

```
State-vector 28 byte, depth reached 15, errors: 1
```

свидетельствует о наличии ошибки. Строки

```
pan: claim violated! (at depth 15)
pan: wrote SWC1.trail
```

показывают, что требование было нарушено и создан *trail*-файл.

Так как была обнаружена ошибка, то выполняется пятый этап верификации автоматной модели. При этом *SPIN* обрабатывает *trail*-файл и выводит отчет, содержащий контрпример (листинг 5).

Листинг 5. Вывод контрпримера верификатором *SPIN*

```
Never claim moves to line 13    [(1)]
              State 0: Waiting for activation
Never claim moves to line 11    [(((lastEvent>10)&&(stateA0==1)))]
              Going to state 1
Never claim moves to line 21    [(1)]
spin: trail ends after 15 steps
#processes: 2
                stateA0 = 1
                lastEvent = 11
15:    proc 1 (Model) line  27 "SWC1" (state 65)
15:    proc 0 (:init:) line 133 "SWC1" (state 2) <valid end state>
15:    proc - (:never:) line  22 "SWC1" (state 16) <valid end state>
2 processes created
```

Из листинга 5 видно, что контрпример содержит путь до состояния 1, и последним входным воздействием было `x11`.

ВЫВОДЫ

Верификация программ, созданных с использованием автоматного подхода, обладает следующими достоинствами:

- автоматическое построение модели для верификации;
- автоматический переход от контрпримера к трассе ошибки на диаграмме состояний автомата;
- автоматная модель для генерации исходного кода и верификации совпадают;
- точное определение проверяемых свойств с помощью формул темпоральной логики;
- возможность верифицировать автоматные модели уже на стадии проектирования ПО;

- использование широко распространенных программ-верификаторов.

Верификатор *SPIN* является мощным свободно распространяемым инструментальным средством и давно используется для верификации программных систем ответственного назначения [11].

Излагаемый в статье материал был рассмотрен на секции «Технология разработки ПО» научно-технического совета ОАО «Концерн «НПО «Аврора» и был рекомендован к опубликованию.

СПИСОК ЛИТЕРАТУРЫ

1. *Кулямин В. В.* Методы верификации программного обеспечения //Институт системного программирования РАН. http://www.ict.edu.ru/ft/005645/62322e1-st09.pdf
2. *Поликарпова Н. И., Шалыто А. А.* Автоматное программирование. СПб.: Питер, 2010.
3. *Ремизов А. О., Шалыто А. А.* Автоматный подход к созданию программного обеспечения БИУС /Состояние, проблемы и перспективы создания корабельных информационно-управляющих комплексов (эффективность, надежность, экономика). ОАО «Моринформсистема-Агат». 2010, с. 155 – 159.
4. *Карпов Ю. Г.* MODEL CHECKING. Верификация параллельных и распределенных программных систем. СПб.: БХВ-Петербург, 2010.
5. *Егоров К. В., Шалыто А. А.* Методика верификации автоматных программ //Информационно-управляющие системы. 2008. № 5, с. 15 – 21.
6. *Вельдер С. Э., Шалыто А. А.* О верификации простых автоматных программ на основе метода Model checking //Информационно-управляющие системы. 2007. № 3, с. 27 – 38.
7. *Лукин М. А., Шалыто А. А.* Верификация автоматных программ с использованием верификатора SPIN. //Научно-технический вестник Санкт-Петербургского государственного университета информационных технологий, механики и оптики. 2008. Вып. 53. Автоматное программирование, с. 145 – 162.
8. *Васильева К. А., Кузьмин Е. В.* Верификация автоматных программ с использованием LTL //Моделирование и анализ информационных систем. 2007. № 1, с. 3 – 14.
9. http://spinroot.com.
10. http://xmlstar.sourceforge.net.
11. *Риган П., Хемилтон С.* NASA: Миссия надежна //Открытые системы. 2004. № 3. http://www.osp.ru/os/2004/03/184060.