

# Автоматное программирование

Хлебников Андрей Александрович

28 августа 2017 г.

# Оглавление

<b>Введение . . . . .</b>	<b>1</b>
<b>Описание объекта управления . . . . .</b>	<b>2</b>
Функциональные требования . . . . .	2
Диаграмма состояний . . . . .	2
Таблица состояний . . . . .	2
Блок-схема . . . . .	2
Функциональная схема . . . . .	2
<b>Верификация программ методом Model Checking . . . . .</b>	<b>3</b>
Системы переходов . . . . .	4
Понятие системы переходов . . . . .	4
Темпоральная логика . . . . .	4
LTL . . . . .	5
CTL . . . . .	5
Язык Promela . . . . .	5
Типы данных . . . . .	6
Процессы . . . . .	6
Атомарные конструкции . . . . .	7
Каналы сообщений . . . . .	7
Ветвления и кнструкции управления . . . . .	8
Циклы . . . . .	9
Безусловные переходы . . . . .	9
Проверки . . . . .	9
Составные типы данных . . . . .	10
Исполняемость . . . . .	10
Ключевые слова . . . . .	11
SPIN . . . . .	13
<b>Автоматное программирование . . . . .</b>	<b>14</b>
<b>Тестирование . . . . .</b>	<b>15</b>
Модульное тестирование . . . . .	15
Ключевые понятия . . . . .	15
Утверждения ( <b>assertion</b> ) . . . . .	16
Запуск тестов . . . . .	17
Флаги . . . . .	17
Интеграционное тестирование . . . . .	17

Эмуляция устройств . . . . .	17
<b>Описание проекта . . . . .</b>	<b>18</b>
CMake . . . . .	18
Makefile . . . . .	18
<b>Практические работы . . . . .</b>	<b>19</b>
Описание объекта управления . . . . .	19
Модель Promela . . . . .	23
Проект Си . . . . .	25
Модульное тестирование . . . . .	27
Обработка ошибок . . . . .	28
Логирование . . . . .	29
Эмулирование внешних устройств . . . . .	30
<b>Лабораторные работы . . . . .</b>	<b>31</b>
Задания . . . . .	31
Требования к оформлению кода . . . . .	32
Отступы . . . . .	32
Объявление переменных . . . . .	32
Пробелы . . . . .	32
Фигурные скобки . . . . .	33
Круглые скобки . . . . .	35
Использование конструкции <code>switch</code> . . . . .	35
Разрыв строк . . . . .	36
Наследование и ключевое слово <code>virtual</code> . . . . .	36
Главное исключение . . . . .	36
Требования к оформлению работы . . . . .	36
<b>Приложение . . . . .</b>	<b>38</b>
Б.П. Кузнецов об автоматном программировании . . . . .	38
Описание опций SPIN . . . . .	40
BNF Языка Promela . . . . .	44
Примеры исходного кода . . . . .	48

# Введение

Автоматное программирование, по сравнению с другими распространенными подходами к разработке сложных программных систем имеет много недостатков Б.П. Кузнецов об автоматном программировании, но также и ряд преимуществ которые будут освещены далее в этом курсе.

Подход к разработке сложных программных систем был построен на основе подхода А. А. Шалыто [2] и состоит из следующих этапов:

1. Создание схемы связей блока управления с объектом управления и системой верхнего уровня.
2. Разработка перечня и описания входных и выходных переменных.
3. Получение алгоритма работы исходя из поставленной задачи.
4. Эвристическое проектирование системы графов переходов конечных автоматов.
5. Описание модели Promela.
6. Верификация модели (с дополнительной проверкой LTL\* соответствия спецификации).
7. Описание проекта, кодирование при помощи языка Си.
8. Написание тестов (эмуляторов устройств) проверки работоспособности периферийных модулей системы.

Существуют также языки автоматного программирования, результатом работы которых является программный код на различных языках, например FSML [6].

# Описание объекта управления

Функциональные требования

Диаграмма состояний

Таблица состояний

Блок-схема

Функциональная схема

# Верификация программ методом Model Checking

Тестирование программы может  
весьма эффективно  
продемонстрировать наличие  
ошибок, но ненадежно  
неадекватно для демонстрации  
их отсутствия

---

Эдсгер Вибе Дейкста

**Model Checking** [3] — это автоматизированный подход, позволяющий для заданной модели поведения системы с конечным (возможно, очень большим) числом состояний и логического свойства (требования) проверить, выполняется ли это свойство в рассматриваемых состояниях данной модели. Алгоритмы для **Model Checking** обычно базируются на полном переборе пространства состояний модели. При этом для каждого состояния проверяется, удовлетворяет ли оно сформулированным требованиям. Алгоритмы гарантированно завершаются, так как модель программы конечна.

## Системы переходов [5]

### Понятие системы переходов

**Системой переходов (СП)** называется пятерка  $S$  вида

$$S = (P, Q, \delta, L, Q^0) \quad (1)$$

компоненты которой имеют следующий смысл.

1.  $P$  - множество, элементы которого называются **утверждениями**
2.  $Q$  - множество, элементы которого называются **состояниями СП**  $S$
3.  $\delta$  - бинарное отношение на  $Q$  (т.е.  $\delta \subseteq Q \times Q$ ) называемое **отношением перехода**
4.  $L$  - функция вида

$$L : Q \times P \rightarrow \{0, 1\} \quad (2)$$

называемая **оценкой**, которая имеет следующий смысл: для каждого  $q \in Q$  и каждого  $p \in P$  утверждение  $p$  считается

- **истинным** в состоянии  $q$ , если  $L(q, p) = 1$ ,
- **ложным** в состоянии  $q$ , если  $L(q, p) = 0$

выражение  $L(q, p)$  может записываться более компактно в виде знакосочетания  $p(\rho)$

5.  $Q^0 \in Q$  - множество **начальных состояний**

## Темпоральная логика

Одним из языков, на котором можно специфицировать свойства систем, является темпоральная логика [5]. Свойства систем описываются в темпоральной логике при помощи темпоральных формул (которые мы будем называть также просто формулами). Примеры свойств, которые могут описываться в темпоральной логике:

1. система при любом варианте своего функционирования не будет находиться ни в одном из состояний из заданного класса
2. система при некотором функционировании когда-нибудь попадёт в некоторое состояние из заданного класса

Как правило, при проведении рассуждений о темпоральных формулах рассматриваются не все возможные формулы, а только формулы из некоторого ограниченного класса. Классы темпоральных формул принято называть темпоральными логиками, или просто логиками, т.е. словосочетание «темпоральная логика» имеет два значения:

- в первом значении – это язык, на котором можно выражать спецификации,
- а во втором – некоторый класс темпоральных формул.

Наиболее известны темпоральные логики

- CTL (Computational Tree Logic), и
- LTL (Linear Temporal Logic).

Во всех темпоральных формулах основными структурными элементами являются утверждения. Утверждения имеют тот же смысл, что и в системах переходов, т.е. для каждого состояния  $q$  каждой СП (Понятие системы переходов) и каждого утверждения  $p$  определено значение  $q(p) \in \{0, 1\}$ . Совокупность всех утверждений обозначается символом  $\rho$ . Каждая темпоральная логика  $\Phi$  должна удовлетворять следующим условиям.

1.  $P \supseteq \Phi$ .
2. Символы 1 и 0 принадлежат  $\Phi$ .
3. Если  $\psi, \eta \in \Phi$ , то знакосочетания

$$\neg\psi, \psi \wedge \eta, \psi \vee \eta \quad (3)$$

тоже принадлежат логике  $\Phi$ .

Формулы (3) называются булевыми комбинациями формул  $\psi$  и  $\eta$ .

## LTL

## CTL

## Язык Promela

PROMELA (Process or Protocol Meta Language) – это язык описания моделей верификации, созданный Gerard J. Holzmann [8]. Язык поддерживает создание процессов для проверки распределенных моделей. Модели в языке могут взаимодействовать между собой при помощи каналов сообщений как в синхронном режиме, так и в асинхронном. Модели описанные при помощи языка могут быть обработаны и проанализированы SPIN о чем будет рассказано в последующих главах. Существуют иные реализации и утилиты использующие язык Promela, но пока они рассматриваться не будут.

В основном, язык предназначен для проверки логики работы параллельных систем. Модели описанные Promela и обработанные утилитой SPAN проверяют модель на корректность в режиме случайной или последовательной симуляции или генерируют код на C для быстрой и полной проверки в системном окружении. В процессе симуляции и проверки SPIN проверяет отсутствия deadlocks<sup>1</sup>,

---

<sup>1</sup><https://ru.wikipedia.org/wiki/Deadlock>



неопределенных состояний и неиспользуемых частей кода. Также данный подход может проверять правильность системных инвариантов<sup>2</sup>, а также поиска зацикливаний и неправильных ветвлений. Также он поддерживает проверку LTL ограничений.

Список спецификаций различных систем, модель которых описана на языке Promela приведена в статье Alberto Lluch<sup>3</sup>

## Типы данных

Имя	Размер(в битах)	Тип	Диапазон значений
bit	1	unsigned	0..1
bool	1	unsigned	0..1
byte	8	unsigned	0..255
mtype	8	unsigned	0..255
short	16	signed	$-2^{15}..2^{15} - 1$
int	32	signed	$-2^{31}..2^{31} - 1$

Типы `bit` и `bool` это синонимы.

Также, переменные, могут быть представлены в виде массива. Пример определения:

```
int x [10];
```

в данном примере определен массив из 10 элементов типа `int` с именем `x`

Доступ к элементам массива осуществляется по индексам, в свою очередь индекс не может превышать размерность массива.

Имена переменных и процессов не должно совпадать с ключевыми словами языка Ключевые слова

## Процессы

Значения переменных или состояние каналов сообщений могут быть изменены только внутри процесса. Поведение процесса описывается декларацией `proctype`. В примере ниже мы определяем процесс `A` с одной переменной `state`

```
proctype A() {
    byte state;
    state = 3;
}
```

`proctype` только определяет процесс, но не запускает его. При инициализации модели запускается только один процесс с именем `init` который должен быть явным образом задан в каждом Promela описании.

Процесс может быть запущен при помощи оператора `run`, который в качестве аргумента принимает имя запускаемого процесса, заданного декларацией `proctype`. Оператор запуска может быть использован в определении процесса, а не только в процессе инициализации `init`, он предназначен для динамического запуска процессов.

<sup>2</sup>[https://en.wikipedia.org/wiki/Invariant\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Invariant_(computer_science))

<sup>3</sup><http://www.albertolluch.com/research/promelamodels>

Процесс завершает свою работу при достижении окончания определения в блоке **proctype**, а также завершает все дочерние(созданные завершаемым процессом) процессы.

Перед декларации **proctype** может стоять квалификатор **active** который сигнализирует об автоматическом запуске процесса. В свою очередь у **active** можно указать квантификатор, который будет задавать количество запускаемых процессов.

```
active   proctype A() { ... }
active [4] proctype B() { ... }
```

в примере выше описан автоматический запуск двух экземпляров процесса B и автоматический запуск процесса A

## Атомарные конструкции

Последовательность выражений можно обернуть фигурными скобками с ключевым словом **atomic**, тем самым обозначить исполнение последовательности одним единым блоком без разделения другими процессами.

```
atomic {
    ...
}
```

## Каналы сообщений

Каналы сообщений необходимы для осуществления межпроцессного взаимодействия. Соответственно, каналы могут быть глобальными и локальными. Например:

```
chan qname = [16] of {short}
```

в примере мы определили буферный канал сообщений размерностью 16 сообщений типа **short**. Выражение

```
qname ! expression;
```

помещает(посылает) значение заданное выражением **expression** в канал с именем **qname**, оно будет помещено в конец очереди канала. Выражение:

```
qname ? msg;
```

получает сообщение из начала очереди и помещает его в переменную **msg**. Канал работает по механизму **FIFO**

Для того, чтобы определить канал сообщений без очереди, следует в качестве размера передать 0. Пример:

```
chan port = [0] of {byte}
```

Подобного рода каналы работают в синхронном режиме, а именно получатель и отправитель ожидают пока получатель или отправитель не завершаь операцию приема или передачи сообщения.

В случае, если канал сообщений будет заполнен(заполнена очередь), то канал себя ведет как синхронный - блокирует операцию. Канал, в один момент времени может работать или на прием или на передачу. Каналы не являются однонаправленными и их можно использовать совместно несколькими процессами получателями и отправителями.

## Ветвления и конструкции управления

Простейшее сравнение двух переменных:

```
if
:: ( a != b ) -> option1
:: ( a == b ) -> option2
fi
```

в примере имеется две исполняемые последовательности, каждая описывается двойным двоеточием `::`. Только одна последовательность будет исполнена в блоке. Последовательность может быть выбрана только если будет исполнено первое выражение. Первое выражение называется защитным.

В примере выше, мы имеем взаимоисключающие выражения - их не должно быть. Если более чем одно из защитных выражений исполнимо, одно из описанных последовательностей будет выбрано. Если все выражения не исполнимы, процесс блокируется, пока хоть одно из них не будет исполнимо

```
if
:: (A == true ) -> option1;
:: (B == true ) -> option2; /* May arrive here also if A==true */
:: else -> fallthrough_option;
fi
```

The consequence of the non-deterministic choice is that, in the example above, if A is true, both choices may be taken. In "traditional" programming, one would understand an if - if - else structure sequentially. Here, the if - double colon - double colon must be understood as "any one being ready" and if none is ready, only then would the else be taken.

```
if
:: value = 3;
:: value = 4;
fi
```

In the example above, value is non-deterministically given the value 3 or 4.

There are two pseudo-statements that can be used as guards: the timeout statement and the else statement. The timeout statement models a special condition that allows a process to abort the waiting for a condition that may never become

true. The else statement can be used as the initial statement of the last option sequence in a selection or iteration statement. The else is only executable if all other options in the same selection are not executable. Also, the else may not be used together with channels.

## Циклы

Для повторения группы выражений применяются циклы. Пример

```
do
  :: count = count + 1
  :: a = b + 2
  :: (count == 0) -> break
od
```

Только одна последовательность может быть исполнена в единицу времени. После завершения исполнения последовательности исполнение повторяется. Нормальное завершение цикла **break** выражение, тем самым передает управление следующей инструкции после блока цикла.

## Безусловные переходы

Другой путь выхода из цикла - **goto** выражение. Для примера перепишем пример выше

```
do
  :: count = count + 1
  :: a = b + 2
  :: (count == 0) -> goto done
od
done:
  skip ;
```

Переход будет осуществлен на метку с именем done сразу после цикла. Сама метка может быть записана только перед выражением. **skip** это пустая инструкция которая не предпринимает никаких действий.

## Проверки

Выжной частью модели описанной языком **Promela** является утверждение

```
assert (any_boolean_condition)
```

выражение всегда исполняется. Если логическое условие верно - то ничего не происходит, иначе - будет воспроизведена ошибка в процессе верификации при помощи SPIN

## Составные типы данных

При помощи определения `typedef` в языке, можно задать составной тип данных, который будет использоваться по заданному ему имени в любой части модели.

```
typedef MyStruct {  
    short  Field1;  
    byte   Field2;  
};
```

Для доступа к полям составного типа данных осуществляется также как и в языке C посредством вызова знака `..` Пример:

```
MyStruct x;  
x.Field1 = 1;
```

в примере, значение поля `Field1` переменной `x` устанавливается значение 1.

## Исполняемость

Исполняемость модели обеспечивает базовые средства языка для моделирования синхронизации процессов.

```
mtype = M_UP, M_DW;  
chan Chan_data_down = [0] of { mtype };  
chan Chan_data_up   = [0] of { mtype };  
proctype P1 ( chan Chan_data_in, Chan_data_out ) {  
    do  
        :: Chan_data_in ? M_UP -> skip ;  
        :: Chan_data_out ! M_DW -> skip ;  
    od ;  
};  
  
proctype P2 ( chan Chan_data_in, Chan_data_out ) {  
    do  
        :: Chan_data_in ? M_DW -> skip ;  
        :: Chan_data_out ! M_UP -> skip ;  
    od ;  
};  
  
init {  
    atomic {  
        run P1 (Chan_data_up,  Chan_data_down);  
        run P2 (Chan_data_down, Chan_data_up);  
    }  
}
```

В примере два процесса P1 и P2 имеют недетерминированный выбор 1 входа во 2 выход. Возможны два варианта выбора из которых только один будет выбран. Повторение будет бесконечным. При этом модель не получит **deadlock**<sup>4</sup>

Когда SPIN анализирует модель он проверяет ее при помощи недетерминированного алгоритма и проверит все возможные ее состояния. Когда симулятор SPIN будет визуализировать возможные не проверенные связи, он будет использовать генератор случайных чисел, для проверки недетерминированных состояний. Следовательно симулятор может не показать плохие пути выполнения (хотя таких путей в примере нет). Это иллюстрирует разницу между проверкой и симуляцией. Также можно генерировать исполняемый код из моделей Promela с использованием **Refinement**<sup>5</sup>

## Ключевые слова

Список ключевых слов используемых в языке

```
active
assert
atomic
bit
bool
break
byte
chan
d_step
D_proctype
do
else
empty
enabled
fi
full
goto
hidden
if
inline
init
int
len
mtype
empty
never
nfull
```

---

<sup>4</sup><https://ru.wikipedia.org/wiki/Deadlock>

<sup>5</sup> Sharma, Asankhaya. A Refinement Calculus for Promela. 2013 18th International Conference on Engineering of Complex Computer Systems, 2013. doi:10.1109/ICECCS.2013.20  
ссылка на реализацию: <https://github.com/codelion/SpinR.git>

od  
of  
pc\_value  
printf  
priority  
prototype  
provided  
run  
short  
skip  
timeout  
typedef  
unless  
unsigned  
xr  
xs

Полное описание языка в форме Бэкуса-Наура представлено в приложении  
BNF Языка Promela

## SPIN

SPIN (англ. **S**imple **P**romela **I**nterpreter)<sup>6</sup> - утилита для верификации корректности распределенных программных моделей. Служит для автоматизированной проверки моделей. Развивается Gerard J. Holzmann и его коллегами из **Unix group** центра **Computing Sciences Research Center** в **Bell Labs** начиная с 1980 года. С 1991 года программа распространяется бесплатно вместе с исходными кодами.

В отличие от многих программ для проверки моделей, SPIN не выполняет работу сам, а генерирует программу на языке Си, которая решает конкретную задачу. За счет этого достигается экономия памяти и повышение производительности, и становится возможным использовать фрагменты кода на языке Си непосредственно из модели. SPIN предоставляет множество опций для ускорения проверки моделей.

Описание опций можно посмотреть в приложении Описание опций SPIN

---

<sup>6</sup>[https://en.wikipedia.org/wiki/SPIN\\_model\\_checker](https://en.wikipedia.org/wiki/SPIN_model_checker)



# Автоматное программирование

Автоматное программирование [1] - это некая общая парадигма программирования, суть которой заключается в том, что создаваемая программа рассматривается как реализация некоторого управляющего автомата.

В традиционном программировании в последнее время все шире используется понятие «событие», тогда как предлагаемый стиль программирования базируется на понятии «состояние». Понятия «состояние» и «входное воздействие», которое может быть входной переменной или событием, в совокупности образуют «автомат без выхода». Добавляя к ним еще понятие «выходного воздействия», получаем «автомат» (конечный, детерминированный) [7].

Особенность этого подхода состоит в том, что при его использовании автоматы задаются графами переходов. Для различения однотипных вершин вводится понятие «кодирование состояний». При выборе «многозначного кодирования» с помощью одной переменной можно различить состояния, число которых совпадает с числом возможных значений выбранной переменной. Это позволило ввести в программирование понятия «наблюдаемость» и «управляемость» программ, широко используемые в теории управления.

В рамках предлагаемого подхода программирование выполняется «через состояния», а не «через переменные» (флаги), что позволяет лучше понять и специфицировать задачу и ее составные части. При этом необходимо отметить, что в автоматически-ориентированном программировании проектирование, реализация и отладка проводятся в терминах автоматов. Благодаря этому в рамках предлагаемого подхода от графа переходов к тексту программы можно переходить формально и изоморфно.

# Тестирование

Если отладка - процесс удаления ошибок, то программирование должно быть процессом их внесения

---

Эдсгер Вибе Дейкста

## Модульное тестирование<sup>7</sup>

Модульное тестирование, или юнит-тестирование (англ. **unit testing**)<sup>8</sup> — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

Цель модульного тестирования — изолировать отдельные части программы и показать, что по отдельности эти части работоспособны.

Существует множество библиотек способствующих к быстрому написанию тестов для языка Си. Мы будем использовать **GoogleTest**<sup>9</sup>.

## Ключевые понятия

Ключевым понятием в **Google test framework** является понятие утверждения (**assert**). Утверждение представляет собой выражение, результатом выполнения которого может быть успех (**success**), некритический отказ (**nonfatal failure**) и критический отказ (**fatal failure**). Критический отказ вызывает завершение выполнения теста, в остальных случаях тест продолжается. Сам тест представляет собой набор утверждений. Кроме того, тесты могут быть сгруппированы в наборы (**test case**). Если сложно настраиваемая группа объектов должна быть использована в различных тестах, можно использовать фиксации (**fixture**). Объединенные наборы тестов являются тестовой программой (**test program**).

---

<sup>7</sup>В подготовке данной части материала использовалась статья **Google testing framework (gtest)**<https://habrahabr.ru/post/119090/>

<sup>8</sup>[https://en.wikipedia.org/wiki/Unit\\_testing](https://en.wikipedia.org/wiki/Unit_testing)

<sup>9</sup><https://github.com/google/googletest.git>

## Утверждения (assertion)

Утверждения, порождающие в случае их ложности критические отказы начинаются с `ASSERT_`, некритические — `EXPECT_`. Следует иметь ввиду, что в случае критического отказа выполняется немедленный возврат из функции, в которой встретилось вызвавшее отказ утверждение. Если за этим утверждением идет какой-то очищающий память код или какие-то другие завершающие процедуры, можете получить утечку памяти.

Имеются следующие утверждения (некритические начинаются не с `ASSERT_`, а с `EXPECT_`):

- Простейшие логические

```
ASSERT_TRUE(condition);  
ASSERT_FALSE(condition);
```

- Сравнение

```
ASSERT_EQ(expected, actual); - =  
ASSERT_NE(val1, val2); - !=  
ASSERT_LT(val1, val2); - <  
ASSERT_LE(val1, val2); - <=  
ASSERT_GT(val1, val2); - >  
ASSERT_GE(val1, val2); - >=
```

- Сравнение строк

```
ASSERT_STREQ(expected_str, actual_str);  
ASSERT_STRNE(str1, str2);  
ASSERT_STRCASEEQ(expected_str, actual_str); - регистронезависимо  
ASSERT_STRCASENE(str1, str2); - регистронезависимо
```

- Проверка на исключения

```
ASSERT_THROW(statement, exception_type);  
ASSERT_ANY_THROW(statement);  
ASSERT_NO_THROW(statement);
```

- Проверка предикатов

```
ASSERT_PREDN(pred, val1, val2, ..., valN); - N <= 5  
ASSERT_PRED_FORMATN(pred_format, val1, val2, ..., valN); - работа
```

- Сравнение чисел с плавающей точкой

```
ASSERT_FLOAT_EQ(expected, actual); - неточное сравнение float
ASSERT_DOUBLE_EQ(expected, actual); - неточное сравнение double
ASSERT_NEAR(val1, val2, abs_error); - разница между val1 и val2 не превышает
```

- Вызов отказа или успеха

```
SUCCEED();
FAIL();
ADD_FAILURE();
ADD_FAILURE_AT("file_path", line_number);
```

## Запуск тестов

Объявив все необходимые тесты, мы можем запустить их с помощью функции `RUN_ALL_TESTS()`. Функцию можно вызывать только один раз. Желательно, чтобы тестовая программа возвращала результат работы функции `RUN_ALL_TESTS()`, так как некоторые автоматические средства тестирования определяют результат выполнения тестовой программы по тому, что она возвращает.

## Флаги

Вызванная перед `RUN_ALL_TESTS()` функция `InitGoogleTest(argc, argv)` делает вашу тестовую программу не просто исполняемым файлом, выводящим на экран результаты тестирования. Это целостное приложение, принимающее на вход параметры, меняющие его поведение. Как обычно ключи `-h`, `-help` дадут вам список всех поддерживаемых параметров. Перечислю некоторые из них (за полным списком можно обратиться к документации).

```
./test --gtest_filter=TestCaseName.*-TestCaseName.SomeTest - запустить все тесты набора
./test --gtest_repeat=1000 --gtest_break_on_failure - запустить тестирующую программу
./test --gtest_output="xml:out.xml" - помимо выдачи в std::out будет создан out.xml -
./test --gtest_shuffle - запускать тесты в случайном порядке
```

Если вы используете какие-то параметры постоянно, можете задать соответствующую переменную окружения и запускать исполняемый файл без параметров. Например задание переменной `GTEST_ALSO_RUN_DISABLED_TESTS` ненулевого значения эквивалентно использованию флага `-gtest_also_run_disabled_tests`.

## Интеграционное тестирование

### Эмуляция устройств

# Описание проекта

## CMake

CMake (от англ. `cross platform make`)<sup>10</sup> — это кроссплатформенная система автоматизации сборки программного обеспечения из исходного кода. CMake не занимается непосредственно сборкой, а лишь генерирует файлы управления сборкой из файлов `CMakeLists.txt`:

1. `Makefile` в системах `Unix` для сборки с помощью `make`;
2. файлы `projects/solutions (.vcxproj/.vcproj/.sln)` в `Windows` для сборки с помощью `Visual C++`;
3. проекты `XCode` в `Mac OS X`

Наиболее полную информацию по использованию CMake можно найти на официальном сайте <https://cmake.org/documentation/>

## Makefile

---

<sup>10</sup><https://en.wikipedia.org/wiki/CMake>

# Практические работы

Каждая практическая работа рассчитана на 1-3 практических занятия и соответствует лабораторным работам.

## Описание объекта управления

Попробуем описать объект управления - "кондиционер". Довольно таки распространенное в бытовом плане устройство, к тому же интуитивно понятно как оно работает. Поэтому не сложно будет выделить основные состояния в которых может находится кондиционер.

Первое с чего можно начать - это включение устройства. Поэтому первичное состояние кондиционера - выключено. Далее мы нажимаем кнопку на пульте управления(обработка сигналов с пульта управления это немного более сложный процесс и для простоты мы будем рассматривать пульт управления как некое абстрактное устройство которое может менять состояние нашего объекта с включенного на выключенный и наоборот) либо на самом кондиционере. После включения питания кондиционер восстанавливает параметры(параметры установленные до выключения питания) и переходит в состояние "Управление". Какие параметры могут быть у нашего кондиционера:

1.  $T$  - температура которую необходимо поддерживать кондиционеру
2.  $W_1$  - Минимальное время нагрева до проверки температуры
3.  $W_2$  - Минимальное охлаждения до проверки температуры
4.  $W_3$  - Минимальное время ожидания до проверки температуры

сразу хочется оговориться, что для простоты работы и понимания функциональные требования к кондиционеру были сокращены до минимальных, а именно до поддержания установленной температуры окружающей среды помещения в актуальном состоянии.

Как видим параметров у нашего объекта не много, при этом параметры  $W_1, W_2, W_3$  являются сервисными параметрами и их пользователи изменять не могут, но они являются важными для описания, поэтому были перечислены.

После восстановления параметров и перехода в состояние "Управление" объект управления должен получить(измерить) температуру окружающей среды и в зависимости от результат сравнения полученного значения температуры  $t$  с температурой которую надо поддерживать  $T$  перейти в соответствующее состояние:

1.  $t < T$  - температура окружающей среды меньше поддерживаемой температуры, поэтому следует перейти в состояние "Нагрев"
2.  $t > T$  - температура окружающей среды больше поддерживаемой температуры, поэтому следует перейти в состояние "Охлаждение"
3.  $t == T$  - температура окружающей среды соответствует поддерживаемой температуры, поэтому следует перейти в состояние "Ожидание"

перейдя в нужное состояние кондиционер либо произведет действия по охлаждению, либо по нагреву, либо ничего делать не будет (будет экономить электроэнергию, также называемый "Режим ожидания"). Из любого из этих состояний объект управления переходит в состояние "Управления" по условию:

1.  $w > W_1$  - внутренний счетчик времени работы в состоянии "Нагрев" больше предельного значения
2.  $w > W_2$  - внутренний счетчик времени работы в состоянии "Охлаждение" больше предельного значения
3.  $w > W_3$  - внутренний счетчик времени работы в состоянии "Ожидание" больше предельного значения

при этом, при вхождении в состояние внутренние счетчики сбрасываются.

После нажатия кнопки выключения на пульте управления либо на самом устройстве, объект управления переходит из состояния "Управление" в состояние "Выключен". Перед выключением мы сохраняем параметры. Как видите, мы можем перейти в состояние "Выключен" только из состояния "Управления" что накладывает некоторые ограничения на нашу модель, а именно ожидает перехода в состояние "Управление". Наша модель не является критичной ко времени срабатывания (если бы мы управляли задвижками ТВЭЛ (тепловыделяющий элемент) в АЭС мы бы в первую очередь задумались о реакции системы) поэтому мы упростили ее.

При помощи PlantUML<sup>11</sup> построим диаграмму состояний.

Для описания диаграммы перейдем на сайт PlantUML <http://www.plantuml.com/plantuml/uml> и наберем следующую последовательность символов (если не хочется вводить, можно перейти по ссылке):

```
@startuml
title "Работа кондиционера"

legend
| =      | = Описание      |
| t      | Температура окружающей среды |
| T      | Граничная температура      |
| w      | Текущее значение таймера     |
| W1     | Таймаут нагрева              |
```

<sup>11</sup><https://en.wikipedia.org/wiki/PlantUML>

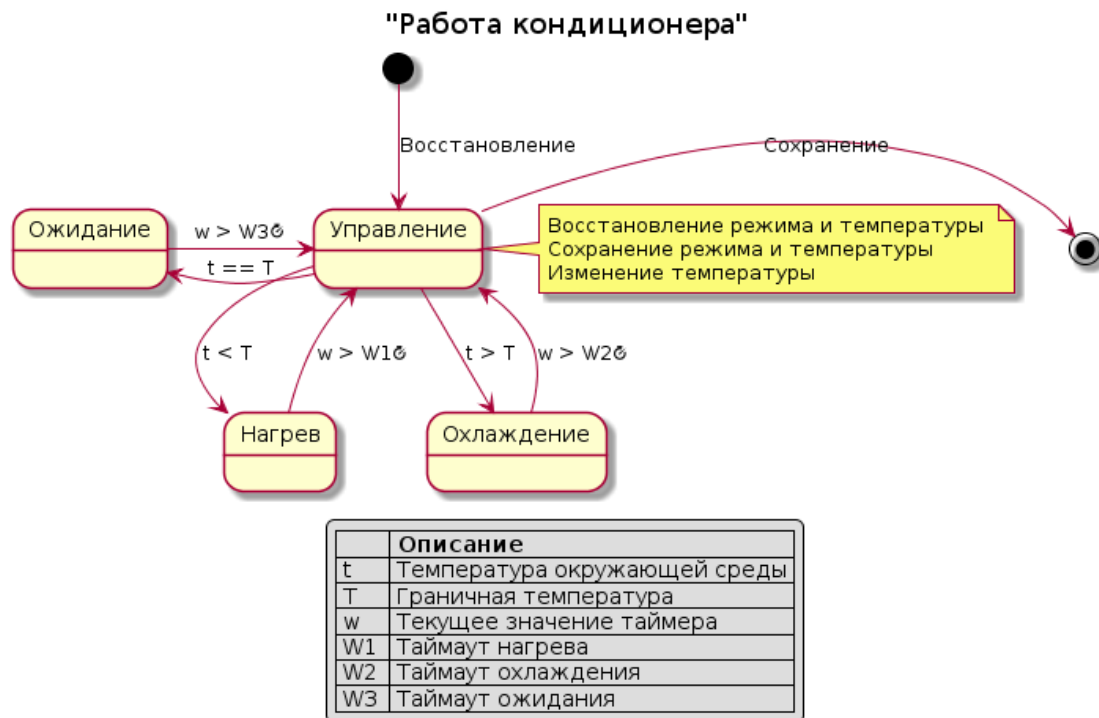


Рис. 1: Диаграмма состояний работы кондиционера

```

| W2 | Таймаут охлаждения |
| W3 | Таймаут ожидания   |
end legend

[*] -down-> Управление: Восстановление
Управление -> [*]: Сохранение
Управление -down-> Нагрев: t < T
Управление -down-> Охлаждение: t > T
Нагрев -up-> Управление: w > W1<&timer>
Охлаждение -up-> Управление: w > W2<&timer>
Управление -left-> Ожидание: t == T
Ожидание -right-> Управление: w > W3<&timer>

```

```

note right of Управление : Восстановление режима и температуры \nСохранение режима и
@enduml

```

После нажатия на клавишу "Submit" мы получим следующую диаграмму:

Давайте более детально рассмотрим текст описания диаграммы. Ключевое слово `@startuml` начинает блок описания диаграммы, а `@enduml` завершает его. `title` задает наименование нашей диаграммы. `legend` начинает блок легенды диаграммы, а `end legend` завершает ее. Внутри легенды приведена конструкция задания таблицы, в которой мы описываем необходимые нам параметры. Состояния начала и окончания задаются при помощи `[*]` последовательности. Стрелочки `->` задают переходы между состояниями, а `:` задают условия переходов. `note`



`right of` добавляет подсказку к состоянию.

Подробности использования языка PlantUML можно найти по ссылке [http://plantuml.com/PlantUML\\_Language\\_Reference\\_Guide.pdf](http://plantuml.com/PlantUML_Language_Reference_Guide.pdf)

## Модель Promela

Описанный выше объект управления формализуем при помощи Язык Promela.

```

mtype = {START, ON, OFF, CONTROL, COOLING, HEATING, WAITING};

mtype state = START;
int guard_temperature = 12;
int current_temperature = 0;
int waiting_count = 10;

proctype conditioner() {
    printf("[Conditioner %d] Вход\n", _pid);
    do
        :: (state == OFF) ->
            printf("[Conditioner %d] Сохраняем температуру и параметры\n", _pid);
            break ;
        :: (state == ON) ->
            printf("[Conditioner %d] Загрузка темепратуры и параметров\n", _pid);
            guard_temperature = 12;
            current_temperature = 14;
            printf("[Conditioner %d] Loaded: %dC, Current: %dC\n",
                _pid, guard_temperature, current_temperature);
            state = CONTROL;
        :: (state == CONTROL) ->
            printf("[Conditioner %d] Управление\n", _pid);
            /* Получаем тепуратуру помещения */
            printf("[Conditioner %d] Температура: %dC\n",
                _pid, current_temperature);
            if
                :: (current_temperature > guard_temperature) ->
                    state = COOLING
                :: (current_temperature < guard_temperature) ->
                    state = HEATING;
                :: else ->
                    state = WAITING;
            fi
        :: (state == COOLING) ->
            assert (current_temperature > guard_temperature);
            printf("[Conditioner %d] Охлаждение\n", _pid);
            current_temperature--;
            state = CONTROL;
        :: (state == HEATING) ->
            assert (current_temperature < guard_temperature);
            printf("[Conditioner %d] Нагрев\n", _pid);
            current_temperature++;
            state = CONTROL;
    od
}

```

```
:: (state == WAITING) ->
    assert (current_temperature == guard_temperature);
    printf("[Conditioner %d] Ожидание\n", _pid);
    state = CONTROL;
    if
        :: (waiting_count == 5) ->
            printf("[Conditioner %d] Эмуляция. Понижение температуры.\n", _pid);
            current_temperature = guard_temperature - 2;
            waiting_count--;
        :: (waiting_count < 0) ->
            state = OFF;
        :: else ->
            waiting_count--;
    fi
od
printf("[Conditioner %d] Выход\n", _pid);
}
```

```
init {
    run conditioner();
    assert (state == START);
    printf("[Init] Запуск\n");
    state = ON
}
```

Определено 7 состояний:

1. START - ?
2. ON - Состояние включение объекта управления
3. OFF - Состояние выключение объекта управления
4. CONTROL - Состояние управления
5. COOLING - Состояние нагрева
6. HEATING - Состояние охлаждения
7. WAITING - Состояние ожидания

## Проект Си

Листинг 1: Пример управления состояниями на С

```
#include "conditioner.h"

int
business_fsm(enum State      state,
             struct ConditionerInter *cond,
             struct StorageInter  *storage) {
    int ret = 1;
    do {
        if (cond->is_running && !cond->is_running(state)) {
            state = _PowerOff;
        }
        switch (state) {
            case _PowerOff: {
                ret = storage->property_store();
                break;
            }
            case _PowerOn: {
                storage->property_load();
                state = _Control;
                break;
            }
            case _Waiting: {
                int wait = storage->property_get_int(WaitingIdle);
                cond->process_engine(Waiting, wait);
                state = _Control;
                break;
            }
            case _Heating: {
                int wait = storage->property_get_int(WaitingHeating);
                cond->process_engine(Heating, wait);
                state = _Control;
                break;
            }
            case _Cooling: {
                int wait = storage->property_get_int(WaitingCooling);
                cond->process_engine(Cooling, wait);
                state = _Control;
                break;
            }
            case _Control: {
                int guard_temperature =
                    storage->property_get_int(Temperature);
                int current_temperature = cond->get_current_temperature();
                if (current_temperature > guard_temperature) {
                    state = _Cooling;
                } else if (current_temperature < guard_temperature) {
                    state = _Heating;
                } else {
                    state = _Waiting;
                }
            }
        }
    } while (ret == 1);
}
```

```
        }  
        break;  
    }  
}  
} while (state != _PowerOff);  
return ret;  
}
```

## Модульное тестирование

## Обработка ошибок

## Логи́рование

Пример вывода лога можно посмотреть в приложении статьи А. А. Шалыто и Н. И. Туккель [4].



## Эмулирование внешних устройств

# Лабораторные работы

1. Описание объекта управления
2. Описание модели Promela
3. Структура проекта
4. Модульное тестирование
5. Обработка ошибок
6. Логирование
7. Эмулирование внешних устройств

## Задания

1. Светофор с индикацией оставшегося времени (без детализации часов реального времени)
2. Грузовой лифт (проверка грузоподъемности)
3. Автомобильный манипулятор
4. Супервизор (управление процессами)
5. Автоматический нагреватель воды
6. Дренажный насос
7. Холодильник
8. Турникет метро
9. Банковский терминал выдачи наличных
10. Парковка
11. СКУД
12. ЧПУ фрезер

Номер работы вычисляется путем взятия номера по порядку вышей записи в журнале старосты по модулю 12 и прибавлением к получившемуся числу 1 ( $(N \bmod 12) + 1$ ).

## Требования к оформлению кода

### Отступы

- \* Для обозначения отступа используйте 4 пробела подряд;
- \* Используйте пробелы, а не табуляцию.

### Объявление переменных

- \* Объявляйте по одной переменной в строке;
- \* Избегайте, если это возможно, коротких и запутанных названий переменных (Например: «a», «rbarr», «nughdeget»);
- \* Односимвольные имена переменных подходят только для итераторов циклов, небольшого локального контекста и временных переменных. В остальных случаях имя переменной должно отражать ее назначение;
- \* Заводите переменные только по мере необходимости:

```
// Wrong
int a, b;
char *c, *d;
// Correct
int height;
int width;
char *name_of_this;
char *name_of_that;
```

- \* Функции и переменные должны именоваться с прописной буквы, а если имя переменной или функции состоит из нескольких слов, то они разделяются нижним подчеркиванием;
- \* Избегайте аббревиатур:

```
// Wrong
short Cntr;
char ITEM_DELIM = '\t';
// Correct
short counter;
char item_delimiter = '\t';
```

- \* Имена классов, составных типов данных всегда начинаются с заглавной буквы.

### Пробелы

- \* Используйте пустые строки для логической группировки операторов, где это возможно;
- \* Всегда используйте одну пустую строку в качестве разделителя;

- \* Всегда используйте один пробел перед фигурной скобкой:

```
// Wrong
if(foo){
}

// Correct
if (foo) {
}
```

- \* Всегда ставьте один пробел после \* или &, если они стоят перед описанием типов. Но никогда не ставьте пробелы после \* или & и именем переменной:

```
char *x;
const Class &my_class;
const char * const y = "hello";
```

- \* Бинарные операции отделяются пробелами с 2-х строн;
- \* После преобразования типов не ставьте пробелов;
- \* Избегайте преобразования типов в стиле C, если ваш код на C++:

- \*

```
// Wrong
char * block_of_memory = (char *) malloc(data.size());
// Correct
char * block_of_memory = reinterpret_cast<char *>( malloc(data.size()) );
```

## Фигурные скобки

- \* Возьмите за основу расстановку открывающих фигурных скобок на одной строке с выражением, которому они предшествуют:

```
// Wrong
if (codec)
{
}

// Correct
if (codec) {
}
```

- \* Исключение: Тело функции и описание класса всегда открывается фигурной скобкой, стоящей на той же строке:

```
static void foo(int g) {
    fprintf(stdout, "foo: %i", g);
}

class Moo {
};
```

- \* Используйте фигурные скобки в условиях, если тело условия в размере превышает одну линию, или тело условия достаточно сложное и выделение скобками действительно необходимо:

```
// Wrong
if (address.is_empty) {
    return false;
}

for (int i = 0; i < 10; ++i) {
    fprintf(stdout, "%i", i);
}

// Correct
if (address.is_empty)
    return true;

for (int i = 0; i < 10; ++i)
    fprintf(stdout, "%i", i);
```

- \* Исключение 1: Используйте скобки, если родительское выражение состоит из нескольких строк или оберток:

```
// Correct
if (address.is_empty || !is_valid()
    || !codec) {
    return false;
}
```

- \* Исключение 2: Используйте фигурные скобки, когда тела ветвлений if-then-else занимают несколько строчек:

```
// Wrong
if (address.is_empty)
    return false;
else {
    fprintf(stdout, "%s", address.c_str());
    ++it;
}

// Correct
if (address.isEmpty()) {
    return false;
} else {
    fprintf(stdout, "%s", address.c_str());
    ++it;
}

// Wrong
if (a)
    if (b)
        ...
    else
```

```
...  
  
// Correct  
// Wrong  
if (a) {  
    if (b)  
        ...  
    else  
        ...  
}
```

- \* Используйте фигурные скобки для обозначения пустого тела условия:

```
// Wrong  
while (a);  
  
// Correct  
while (a) {}
```

## Круглые скобки

- \* Используйте круглые скобки для группировки выражений:

```
// Wrong  
if (a && b || c)  
  
// Correct  
if ((a && b) || c)  
  
// Wrong  
a + b & c  
// Correct  
(a + b) & c
```

## Использование конструкции switch

- \* Операторы case должны быть в одном столбце со switch
- \* Каждый оператор case должен иметь закрывающий break (или return) или комментарий, котрой предполагает намеренное отсутствие break или return:

```
switch (my_enum) {  
case Value1:  
    do_somthing();  
    break;  
case Value2:  
    do_somthing_else();  
    // continue  
default:  
    default_handling();  
    break;
```

```
}
```

## Разрыв строк

- \* Длина строки кода не должна превышать 100 символов. Если надо – используйте разрыв строки.
- \* Запятые помещаются в конец разорванной линии; операторы помещаются в начало новой строки. В зависимости от используемой вами IDE, оператор на конце разорванной строки можно проглядеть:

```
// Correct
if (long_expression
    + other_long_expression
    + other_other_long_expression) {
}

// Wrong
if (long_expression +
    other_long_expression +
    other_other_long_expression) {
}
```

## Наследование и ключевое слово `virtual`

- \* При переопределении `virtual`-метода, *ни за что* не помещайте слово `virtual` в заголовочный файл.

## Главное исключение

- \* Не бойтесь нарушать описанные выше правила, если вам кажется, что они только запутают ваш код.

## Требования к оформлению работы

Лабораторная работа должна состоять из следующих необходимых компонент:

- 1 лист Титульный лист
- 2 лист Описание объекта управления(цели и задачи объекта управления, функциональные требования, граничные условия, параметры и их назначение). Пример Описание объекта управления.
- 3 лист Модель `Promela`. Пример Модель `Promela`.
- 4 лист Диаграмма переходов. Пример Описание объекта управления.
- 5 лист Описание модулей, компонент их взаимодействие.

Работа считается принятой если выполнены условия:

1. Лабораторная работа представлена в бумажном виде и состоит, как минимум, из описанных выше компонент.
2. На электронном носителе или в репозитории(**GitHub**) присутствуют в электронном виде:
  - (a) Исходный код работы выполненный при помощи языка Си и оформленный в соответствии с требованиями Требования к оформлению кода
  - (b) Текст лабораторной работы в формате **doc** или **tex**
  - (c) Собранный исполняемый модуль приложения
  - (d) Проект вашей работы(файлы сборки - **cmake**, необходимые библиотеки - **googletest** как минимум, и т.д.)
3. Ваша работа собирается на тестовом стенде, проходит верификацию модели, проходят все тесты(исполняется тестовый пример)



# Приложение

## Б.П. Кузнецов об автоматном программировании

Б. П. Кузнецов<sup>12</sup>

Автоматное программирование ничуть не лучше любого другого ни по числу допускаемых ошибок, ни по срокам трудоемкости создания и отладки программ. В этом я убедился и в период активного использования как табличного, так и спискового (Switch, Любченко, Зюбин) задания автоматов в программах (к месту и не к месту) и в последующий период «безавтоматного программирования» с 2003 г. по нынешний день (да и до этого, когда «вспомнил» об автоматах). Более того, автоматное программирование доступно отнюдь не большинству, менее понятно и более трудоемко.

Перечислю типичные ошибки, сопровождающие автоматное программирование, как из собственного, так и заимствованного опыта, только лишь на примере составления диаграммы состояний конечного автомата:

1. Не учтенные состояния автомата, вызванные незнанием предметной области;
2. Дублирование (избыточность) состояний, приводящее к непредсказуемому поведению программы;
3. Не учтенные переходы;
4. Лишние переходы;
5. Неверно ориентированные переходы;
6. Неортогональность входного алфавита;
7. Неверное назначение приоритетов переходов при неортогональном алфавите;
8. Не учтенные входные воздействия и неполнота входного алфавита;
9. Не полный учет букв входного алфавита;
10. Путаница, связанная с неверным отождествлением входных воздействий и буквами входного алфавита (наиболее распространенная ошибка);
11. Неверные булевы формулы, отождествляемые с буквами входного алфавита и помечающие переходы (см. п.6);

---

<sup>12</sup>к.т.н Концерн «НПО «Аврора», г. Санкт-Петербург

12. Не учтенные выходные воздействия, в особенности как реакция на ошибочное поведение управляемого объекта и самого управляющего автомата и связи управляющего и операционного автоматов с объектом управления;
13. Неверная пометка состояний и переходов буквами выходного алфавита;
14. Неверное отождествление букв выходного алфавита с выходными воздействиями;
15. Забытое обнуление или продление выходного воздействия;
16. Путаница в отметке переходов и состояний буквами выходного алфавита при использовании совмещенной модели (Мили и Мура) автоматов;
17. Не прослеживаются полные пути в диаграмме состояний;
18. Всевозможные ошибки в полных путях при использовании обратных связей в диаграмме состояний
19. Другие ошибки.

Только одного этого перечня вполне достаточно, чтобы скомпрометировать «непорочность» автоматного программирования. И его мнимые достоинства связаны с тем, что «каждый кулик хвалит свое болото», чем и я в свое время безапелляционно сообщал научному и инженерному сообществам в своих публикациях (см., например, мою статью «Психология автоматного программирования»).

PS. Мне кажется, что приведенный длинный перечень возможных ошибок только подтверждает такое достоинство автоматного программирования как формализация задания логики программы. Все изложенное можно проверять (автоматически и вручную), повышая качество программы. Интересно, как бы выглядел этот перечень для программ, которые пишутся традиционно? Мне кажется, что была бы одна строчка – в логике программы могут быть ошибки... Что с этим делать? А то, что автоматные программы, в отличие написанных иначе, удобно верифицировать методом Model Checking – это разве не достоинство. Графы переходов можно обсуждать с Заказчиками, а программы нельзя. И т. д., и т. п. А.А. Шалыто

## Описание опций SPIN

## Run Time Options for PAN

<b>-A</b>	suppress the reporting of assertion violations (see also <b>-E</b> )
<b>-a</b>	find acceptance cycles (available if compiled <i>without</i> <b>-DNP</b> )
<b>-b</b>	bounded search mode, makes it an error to exceed the search depth, triggering and error trail
<b>-cN</b>	stop at <i>N</i> th error (defaults to first error if <i>N</i> is absent)
<b>-d</b>	print state tables and stop
<b>-E</b>	suppress the reporting of invalid endstate errors (see also <b>-A</b> )
<b>-e</b>	create trails for all errors encountered (default is first one only)
<b>-f</b>	add weak fairness (to <b>-a</b> or <b>-l</b> )
<b>-hN</b>	choose another hash-function, with <i>N</i> : 1..32 (defaults to 1)
<b>-i</b>	search for shortest path to error (causes an increase of complexity)
<b>-l</b>	like <b>-i</b> , but approximate and faster
<b>-J</b>	reverse the evaluation order of nested unless statements (to conform to the one used in Java)
<b>-kN</b>	set the number of hashfunctions used in bitstate hashing mode to <i>N</i> (requires compilation with <b>-DBITSTATE</b> ) The default is <i>k</i> =2. This option was introduced in version 4.2.0.
<b>-l</b>	find non-progress cycles (requires compilation with <b>-DNP</b> )
<b>-mN</b>	set max search depth to <i>N</i> steps (default <i>N</i> =10000)
<b>-n</b>	no listing of unreached states at the end of the run
<b>-q</b>	require empty channels in valid endstates
<b>-r, -P, -C</b>	play back error trail with embedded C code statements
<b>-s</b>	use 1-bit hashing (default is 2-bit hashing, assumes compilation <b>-DBITSTATE</b> ). In version 4.2.0 and later, the option <b>-s</b> is equivalent to <b>-k1</b> .
<b>-V</b>	print <i>Spin</i> version number and stop
<b>-wN</b>	use ahashtable of $2^N$ entries (defaults to <b>-w18</b> )

## Compile Time Options for PAN

### Directives Supported by Xspin

BITSTATE	use supertrace/bitstate instead of exhaustive exploration
MEMCNT=N	set upperbound to the amount of memory that can be allocated usage, e.g.: <b>-DMEMCNT=20</b> for a maximum of $2^{20}$ bytes
MEMLIM=N	set upperbound to the true number of Megabytes that can be allocated; usage, e.g.: <b>-DMEMLIM=200</b> for a maximum of 200 Megabytes (meant to be a simple alternative to MEMCNT)
NOCLAIM	exclude the never claim from the verification, if present
NOFAIR	disable the code for weak-fairness (is faster)
NOREDUCE	disables the partial order reduction algorithm
NP	enable non-progress cycle detection (option <b>-l</b> ), replacing option <b>-a</b> for acceptance cycle detection
PEG	add complexity profiling (transition counts)
SAFETY	optimize for the case where no cycle detection is needed (faster, uses less memory, disables both <b>-l</b> and <b>-a</b> )
VAR_RANGES	compute the effective value range of variables (restricted to the interval 0..255)
CHECK	generate debugging information (see also <b>DEBUG</b> )

### Directives Related to Partial Order Reduction

CTL	allow only those reductions that are consistent with branching time logics like CTL (i.e., the persistent set contains either one or all transitions)
GLOB_ALPHA	consider process death a global action (for compatibility with versions of Spin between 2.8.5 and 2.9.7)
NIBIS	apply a small optimization of partial order reduction (sometimes faster, sometimes not...)
NOREDUCE	disables the partial order reduction algorithm
XUSAFE	disable validity checks of <i>x[rs]</i> assertions (faster, and sometimes useful if the check is too strict, e.g. when channels are passed around as process parameters)

### Directives to Increase Speed

NOBOUNDCHECK	don't check array bound violations (faster)
NOCOMP	don't compress states with fullstate storage (faster, but not compatible with liveness unless <b>-DBITSTATE</b> )
NOFAIR	disable the code for weak-fairness (is faster)
NOSTUTTER	disable stuttering rules (warning: changes semantics) stuttering rules are the standard way to extend a finite execution sequence into and infinite one, to allow for a consistent interpretation of Büchi acceptance rules
SAFETY	optimize for the case where no cycle detection is needed (faster, uses less memory, disables both <b>-l</b> and <b>-a</b> )

## Directives to Reduce Memory Use

BITSTATE	use supertrace/bitstate instead of exhaustive exploration
HC	a state vector compression mode; collapses state vector sizes down to 32+16 bits and stores them in conventional hash-table (a version of Wolper's hash-compact method -- new in version 3.2.2.) Variations: HC0, HC1, HC2, HC3 for 32, 40, 48, or 56 bits respectively. The default is equivalent to HC2.
COLLAPSE	a state vector compression mode; collapses state vector sizes by up to 80% to 90% (see Spin97 workshop paper) variations: add -DSEPPQS or -DJOINPROCS (off by default)
MA=N	use a minimized DFA encoding for the state space, similar to a BDD, assuming a maximum of N bytes in the state-vector (this can be combined with -DCOLLAPSE for greater effect in cases when the original state vector is long)
MEMCNT=N	set upperbound to the amount of memory that can be allocated usage, e.g.: -DMEMCNT=20 for a maximum of 2^20 bytes
MEMLIM=N	set upperbound to the true number of Megabytes that can be allocated; usage, e.g.: -DMEMLIM=200 for a maximum of 200 Megabytes (meant to be a simple alternative to MEMCNT)
SC	enables stack cycling. this will swap parts of a very long search stack to a diskfile during verifications. the runtime flag -m for setting the size of the search stack still remains, but now sets the size of the part of the stack that remains in core. it is meant for rare applications where the search stack is many millions of states deep and eats up the majority of the memory requirements.

## Directives Reserved for Use When Prompted by PAN

NFAIR=N	allocates memory for enforcing weak fairness usage, e.g.: -DNFAIR=3 (default is 2)
VECTORSZ=N	allocates memory (in bytes) for state vector usage, e.g.: -DVECTORSZ=2048 (default is 1024)

## Directives for Debugging PAN Verifiers

VERBOSE	adds elaborate debugging printouts
CHECK	more frugal debugging printouts
SVDUMP	if defined, adds an option -pN to the runtime verifiers to produce a file sv_dump at the end of the run, with a binary representation of all states, using a fixed size of N bytes per state. (see also SDUMP below)
SDUMP	if used in addition to CHECK: adds ascii dumps of state vectors to verbose output (i.e., an ascii version of SVDUMP)

## Directives for Experimental Use

BCOMP	when in BITSTATE mode, this computes hash functions over the compressed state-vector (compressed with byte-masking) in some cases, this can improve the coverage
COVEST	no longer supported, see NOCOVEST
HYBRID_HASH	no longer supported
LC	to be used in combination with BITSTATE hashing only. it is automatically enabled when -DSC is used in BITSTATE mode. LC forces the use of hashcompact compression for stackstates (instead of the default which is full-state storage for states while they are on the search stack, even in bitstate mode). it slows down the search, but can save memory. it uses 4 bytes per state (giving very low probability of collision).
NOCOVEST	omits the coverage estimate that is generated at the end of BITSTATE runs.
NOVSZ	risky - removes 4 bytes from state vector - its length field. in most cases this is redundant - so when memory is tight in fullstate storage, try this mode. if the number of states stored changes when -DNOVSZ is used, the information wasn't redundant... (safety checks will still be valid, but liveness checks may then fail) NOVSZ cannot be combined with COLLAPSE
PRINTF	enables printf during verification runs (Version 2.8 and later -- earlier versions always left these enabled)
RANDSTORE	when in BITSTATE mode, use for instance -DRANDSTORE=33 to reduce the probability of storing the bits in the hasharray to 33%. the value assigned must be between 0 and 99 low values increase the amount of work done (time complexity) and increase the effective coverage for large state spaces. most useful in sequential bitstate hashing runs to improve the accumulative coverage of all runs significantly
REACH	guarantee absence of errors within the -m depth-limit (described in more detail in Newsletter 4 and in the V2.Updates notes for Version 2.2.)
W_XPT=N	in combination with MA, write checkpoint files every multiple of N states stored
R_XPT	in combination with MA, restart a verification run from the last checkpoint file written, can be combined with W_XPT

## Compile Time Options for SPIN

NXT	if defined, the NEXT operator X can be used in LTL formulae; risky, not compatible with partial order reductions
PC	required when compiling Spin on a PC
PRINTF	if defined, printf statements in the model are enabled during the verification process (not recommended)
SOLARIS	required when compiling Spin on a Solaris system

## Run Time Options for SPIN

### Simulation

<b>-B</b>	Suppresses the verbose printout at the end of a simulation run (giving process states etc.).
<b>-b</b>	Suppresses the execution of printf statements within the model (see also -T).
<b>-c</b>	Produce an ASCII approximation of a message sequence chart for a random or guided (when combined with -t) simulation run. See also option -M.
<b>-g</b>	Shows at each time step the current value of global variables (see also -w).
<b>-i</b>	Perform an interactive simulation, prompting the user at every execution step that requires a nondeterministic choice to be made. The simulation proceeds without user intervention when execution is deterministic.
<b>-jN</b>	Skip the first N steps of a random or guided simulation. (See also option -uN.)
<b>-l</b>	In combination with option -p, shows the current value of local variables of the process (see also -w).
<b>-M</b>	Produce a message sequence chart in Postscript form for a random simulation or a guided simulation (when combined with -t), for the model in file , and write the result into file.ps . See also option -c.
<b>-nN</b>	Set the seed for a random simulation to the integer value N. There is no space between the -n and the integer N. -p, shows the current value of local variables of the process.
<b>-p</b>	Shows at each simulation step which process changed state, and what source statement was executed.
<b>-qN</b>	In columnated output (option -c) and elsewhere, suppress the printing of output for send or receive operations on the channel numbered N.
<b>-r</b>	Shows all message-receive events, giving the name and number of the receiving process and the corresponding the source line number. For each message parameter, show the message type and the message channel number and name.
<b>-s</b>	Shows all message-send events.
<b>-T</b>	Suppress the default indentation of output from print statements. By default the output from process i is indented by i spaces. See also option -b.
<b>-t[N]</b>	Perform a guided simulation, following the error trail that was produces by an earlier verification run, see the online manuals for the details on verification. If an optional number is attached (no space between the number and the -t) the error trail with that sequence number is opened, instead of the default trail, without sequence number.
<b>-uN</b>	Stop a random or guided simulation after the first N steps. (See also option -jN.)
<b>-v</b>	Verbose mode, adds some more detail, and generates more hints and warnings about the model.
<b>-w</b>	Even more verbose output with options -l and -g (e.g., prints all variable values, not just those that change).

### Verification Generation

<b>-a</b>	Generate a verifier (model checker) for the specification. The output is written into a set of C files, named pan.[ cbhmt ] that can be compiled, (e.g., cc pan.c ) to produce an executable verifier. The online Spin manuals (see below) contain the details on compilation and use of the verifiers.
<b>-A</b>	Perform property-based slicing, warning the user of all statements and data objects that are likely to be redundant for the stated properties (i.e., in assertions and never claims).
<b>-d</b>	Produce symbol table information for the model specified in file . For each Promela object this information includes the type, name and number of elements (if declared as an array), the initial value (if a data object) or size (if a message channel), the scope (global or local), and whether the object is declared as a variable or as a parameter. For message channels, the data types of the message fields are listed. For structure variables, the 3rd field defines the name of the structure declaration that contains the variable.
<b>-F</b>	file This behaves identical to option -f but will read the formula from the file instead of from the command line. The file should contain the formula as the first line. Any text that follows this first line is ignored, so it can be used to store comments or annotation on the formula. (On some systems the quoting conventions of the shell complicate the use of option -f. Option -F is meant to solve those problems.)
<b>-f</b>	LTL Translate the LTL formula LTL into a never claim. This option reads a formula in LTL syntax from the second argument and translates it into Promela syntax (a never claim, which is Promela's equivalent of a Buchi Automaton). The LTL operators are written: [] (always), <> (eventually), and U (strong until). There is no X (next) operator, to secure compatibility with the partial order reduction rules that are applied during the verification process. If the formula contains spaces, it should be quoted to form a single argument to the Spin command. As the name suggests, a Spin never claim is used to specify behavior than is required to be impossible, i.e., behavior that would violate a user-specified property. This means that to check for compliance with an LTL formula, the formula must be negated explicitly before it is converted into a never claim. Negating an LTL formula is easy: just place the formula "f" in parentheses and negate it: "!f".
<b>-J</b>	Reverse the evaluation order of nested 'unless' statements (so that it conforms to the evaluation order of nested 'catch' statements in Java).
<b>-m</b>	Changes the semantics of send events. Ordinarily, a send action will be (blocked) if the target message buffer is full. With this option a message sent to a full buffer is lost.
<b>-V</b>	Prints the Spin version number and exits.

## BNF Языка Promela

```

spec : module [ module ] *

module : proctype /* proctype declaration */
| init /* init process */
| never /* never claim */
| trace /* event trace */
| utype /* user defined types */
| mtype /* mtype declaration */
| decl_lst /* global vars, chans */

proctype: [ active ] PROCTYPE name '(' [ decl_lst ] ')'
        [ priority ] [ enabler ] '{' sequence '}'

init : INIT [ priority ] '{' sequence '}'

never : NEVER '{' sequence '}'

trace : TRACE '{' sequence '}'

utype : TYPEDEF name '{' decl_lst '}'

mtype : MTYPE [ '=' ] '{' name [ ',' name ] * '}'

decl_lst: one_decl [ ';' one_decl ] *

one_decl: [ visible ] typename ivar [ ',' ivar ] *

typename: BIT | BOOL | BYTE | SHORT | INT | MTYPE | CHAN
| uname /* user defined type names (see utype) */

active : ACTIVE [ '[' const ']' ] /* instantiation */

priority: PRIORITY const /* simulation priority */

enabler : PROVIDED '(' expr ')' /* execution constraint */

visible : HIDDEN | SHOW

sequence: step [ ';' step ] *

step : stmt [ UNLESS stmt ]
| decl_lst
| XR varref [ ',' varref ] *
| XS varref [ ',' varref ] *

```

```

ivar      : name [ '[' const ']' ] [ '=' any_expr | '=' ch_init ]

ch_init   : '[' const ']' OF '{' typename [ ',', typename ] * '}'

varref    : name [ '[' any_expr ']' ] [ '.' varref ]

send      : varref '!' send_args /* normal fifo send */
| varref '!' '!' send_args /* sorted send */

receive   : varref '?' recv_args /* normal receive */
| varref '?' '?' recv_args /* random receive */
| varref '?' '<' recv_args '>' /* poll with side-effect */
| varref '?' '?' '<' recv_args '>' /* ditto */

poll      : varref '?' '[' recv_args ']' /* poll without side-effect */
| varref '?' '?' '[' recv_args ']' /* ditto */

send_args: arg_lst | any_expr '(' arg_lst ')'

arg_lst   : any_expr [ ',', any_expr ] *

recv_args: recv_arg [ ',', recv_arg ] * | recv_arg '(' recv_args ')

recv_arg  : varref | EVAL '(' varref ')' | [ '-' ] const

assign    : varref '=' any_expr /* standard assignment */
| varref '+' '+' /* increment */
| varref '-' '-' /* decrement */

stmtnt    : IF options FI /* selection */
| DO options OD /* iteration */
| FOR '(' range ')' '{' sequence '}' /* iteration */
| ATOMIC '{' sequence '}' /* atomic sequence */
| D_STEP '{' sequence '}' /* deterministic atomic */
| SELECT '(' range ')' /* non-deterministic value selection */
| '{' sequence '}' /* normal sequence */
| send
| receive
| assign
| ELSE /* used inside options */
| BREAK /* used inside iterations */
| GOTO name
| name ':' stmtnt /* labeled statement */
| PRINT '(' string [ ',', arg_lst ] ')'
| ASSERT expr

```



```

| expr /* condition */
| c_code '{' ... '}' /* embedded C code */
| c_expr '{' ... '}'
| c_decl '{' ... '}'
| c_track '{' ... '}'
| c_state '{' ... '}'

range : varref ':' expr '..' expr
| varref IN varref

options : ':' ':' sequence [ ':' ':' sequence ] *

andor : '&' '&' | '|' '|'

binarop : '+' | '-' | '*' | '/' | '%' | '&' | '^' | '|',
| '>' | '<' | '>' '=' | '<' '=' | '=' '=' | '!' '=',
| '<' '<' | '>' '>' | andor

unarop : '~' | '-' | '!'

any_expr: '(' any_expr ')'
| any_expr binarop any_expr
| unarop any_expr
| '(' any_expr '-' '>' any_expr ':' any_expr ')'
| LEN '(' varref ')' /* nr of messages in chan */
| poll
| varref
| const
| TIMEOUT
| NP_ /* non-progress system state */
| ENABLED '(' any_expr ')' /* refers to a pid */
| PC_VALUE '(' any_expr ')' /* refers to a pid */
| name '[' any_expr ']' '@' name /* refers to a pid */
| RUN name '(' [ arg_lst ] ')' [ priority ]
| get_priority( expr ) /* expr refers to a pid */
| set_priority( expr , expr ) /* first expr refers to a pid */

expr : any_expr
| '(' expr ')'
| expr andor expr
| chanpoll '(' varref ')' /* may not be negated */

chanpoll: FULL | EMPTY | NFULL | NEMPTY

string : '"' [ any_ascii_char ] * '"'

```

uname : name

name : alpha [ alpha | number ] \*

const : TRUE | FALSE | SKIP | number [ number ] \*

alpha : 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j'  
| 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't'  
| 'u' | 'v' | 'w' | 'x' | 'y' | 'z'  
| 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J'  
| 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T'  
| 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'  
| '\_'

number : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

## Примеры исходного кода

Листинг 2: Пример C

```
1 printf("Hello")
```

# Литература

- [1] В. Э. Карпов. Автоматное программирование и робототехника.
- [2] А. А. Шалыто and Ю. Ю. Янкин. Автоматное программирование плиз в задачах управления эл...
- [3] А. А. Шалыто and С. Э. Вельдер. О ВЕРИФИКАЦИИ ПРОСТЫХ АВТОМАТНЫХ ПРОГРАММ. УДК 681.3.06.
- [4] А. А. Шалыто and Н. И. Туккель. Switch-ТЕХНОЛОГИЯ — АВТОМАТНЫЙ ПОДХОД К СОЗДАНИЮ ПРОГРАММ. 2000.
- [5] А. М. Миронов. Верификация методом model checking.
- [6] И. А. Лагунов. ТЕКСТОВЫЙ ЯЗЫК АВТОМАТНОГО ПРОГРАММИРОВАНИЯ fsm1 ДЛЯ ИСПОЛНЕНИЯ ПРОГРАММ. 2000. УДК 004.432.4.
- [7] А. А. Шалыто. АВТОМАТНОЕ ПРОГРАММИРОВАНИЕ, 2006.  
При поддержке Министерства образования и науки Российской Федерации в рамках научно-исследовательского проекта...
- [8] Unknown. Promela. <https://en.wikipedia.org/wiki/Promela>.

# Список иллюстраций

1	Диаграммам состояний работы кондиционера . . . . .	21
---	--	----