Математические методы верификации схем и программ

Лекторы:

Захаров Владимир Анатольевич

Подымов Владислав Васильевич

e-mail рассказчика:

valdus@yandex.ru

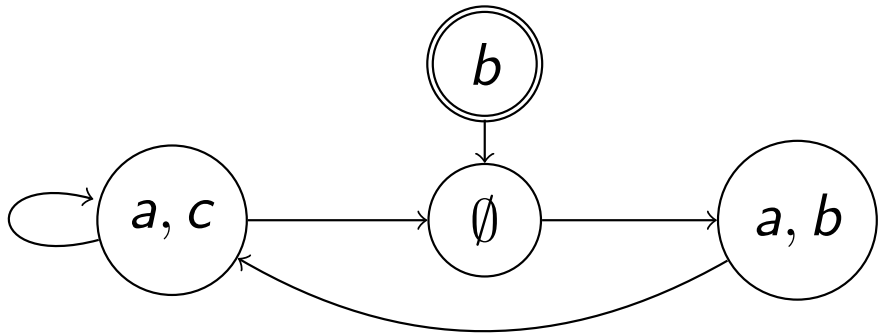
Осень 2016

Семинар 8

SPIN: практика

Упражнение 1: реализация модели Крипке

Проверить выполнимость свойств в модели Крипке



- $\mathbf{G}(a \to b \vee c)$
- $\mathbf{GF}a$
- $\mathbf{FG}a$
- $\mathbf{GF}\neg c \to \mathbf{F}(a \wedge b)$
- $\mathbf{GF}\neg c \to \mathbf{G}(\neg b \mathbf{U} a \wedge b)$

Ещё немного синтаксиса и семантики

У модели на языке PROMELA есть строгая семантика пошаговой работы, и к ней нужно привыкнуть

Однако если в требовании не встречается оператора **X**, некоторые детали семантики можно не принимать во внимание:

$$\cdots \longrightarrow \emptyset \longrightarrow a, b \to \cdots$$

$$\cdots \longrightarrow \emptyset \longrightarrow \emptyset \longrightarrow a, b \to \cdots$$

(*потратили шаг работы на проверку условия*)

$$\cdots \longrightarrow \emptyset \longrightarrow a \longrightarrow a, b \to \cdots$$

(*изменяли a и b на разных шагах*)

- если состояния трассы **дублируются**, то свойства исходной и полученной трасс **одинаковы**, если запрещено использовать оператор **X**
- если в трассу добавляются **существенно новые состояния**, то свойства исходной и полученной трасс могут отличаться

Ещё немного синтаксиса и семантики

В моделях, подаваемых на вход SPIN, можно использовать ряд не упомянутых ранее возможностей языка C/C++

Например:

```
#define N 3
#define good (a == 1) // внимание: скобки!
#define bad (a == 2)
...
active [N] proctype P() { ...
  :: good -> a = 2;
...
}
ltl f {[]!bad}
```

Упражнение 2: доступ в критическую секцию

Два процесса с разделяемой булевой переменной

```
bool free = true;
```

исполняют одну и ту же программу:

```
while(true) {
  NONCRITICAL
  block(free);
  free = false;
  CRITICAL
  free = true;
}
```

Процесс может сколь угодно долго находиться в критической и некритической секциях (CRITICAL и NONCRITICAL соответственно)

Переменная `free` не изменяется процессом, находящимся в этих секциях

Упражнение 2: доступ в критическую секцию

Два процесса с разделяемой булевой переменной

```
bool free = true;
```

исполняют одну и ту же программу:

```
while(true) {
  NONCRITICAL
  block(free);
  free = false;
  CRITICAL
  free = true;
}
```

Инструкция в каждой отдельной строке выполняется атомарно

Инструкция `block(free)` блокирует процесс, пока не станет истинным условие `free`

Упражнение 2: доступ в критическую секцию

Два процесса с разделяемой булевой переменной

```
bool free = true;
```

исполняют одну и ту же программу:

```
while(true) {
  NONCRITICAL
  block(free);
  free = false;
  CRITICAL
  free = true;
}
```

Выяснить,

- могут ли процессы одновременно находиться в своих критических секциях
- возможна ли блокировка обоих процессов сразу,
- верно ли, что, выйдя из некритической секции, процесс рано или поздно достигнет критической

Упражнение 2: доступ в критическую секцию

Два процесса с разделяемой булевой переменной

```
bool free = true;
```

исполняют одну и ту же программу:

```
while(true) {
  NONCRITICAL
  block(free);
  free = false;
  CRITICAL
  free = true;
}
```

Добавить в модель справедливость: процесс не может бесконечно долго находиться в критической секции

Изменить модель так, чтобы пара инструкций `block(free);` `free = false;` выполнялась атомарно: если `block(free)` не блокирует процесс, то немедленно выполнить `free = false`

Упражнение 2: доступ в критическую секцию

Два процесса с разделяемой булевой переменной

```
bool free = true;
```

исполняют одну и ту же программу:

```
while(true) {
  NONCRITICAL
  block(free);
  free = false;
  CRITICAL
  free = true;
}
```

Внимательно посмотреть на флаг “слабая справедливость” в настройках верификации

Ещё немного синтаксиса и семантики

Наряду с обычными булевыми выражениями в LTL-требованиях языка PROMELA можно использовать такие булевы выражения:

```
P[i]@label
```

Здесь

- `P` — имя типа процесса
- `i` — идентификатор процесса
- `label` — метка состояния процесса

Ещё немного синтаксиса и семантики

Наряду с обычными булевыми выражениями в LTL-требованиях языка PROMELA можно использовать такие булевы выражения:

```
P[i]@label
```

У каждого процесса есть свой идентификатор — неотрицательное целое число:

- первому процессу, появившемуся в системе, присваивается идентификатор 0

Ещё немного синтаксиса и семантики

Наряду с обычными булевыми выражениями в LTL-требованиях языка PROMELA можно использовать такие булевы выражения:

```
P[i]@label
```

У каждого процесса есть свой идентификатор — неотрицательное целое число:

- каждому следующему процессу, появившемуся в системе, *в большинстве случаев* присваивается идентификатор — число, следующее за последним присвоенным идентификатором
  - меньшинство случаев — это только те случаи, в которых процесс завершается
  - про эти случаи читайте в документации

Ещё немного синтаксиса и семантики

Наряду с обычными булевыми выражениями в LTL-требованиях языка PROMELA можно использовать такие булевы выражения:

```
P[i]@label
```

У каждого процесса есть свой идентификатор — неотрицательное целое число:

- процессы, запущенные с помощью `active`, появляются в системе в том порядке, в котором встречается слово `active` в тексте модели

Ещё немного синтаксиса и семантики

Системы с `goto` — это плохо, но не всегда, и в синтаксисе PROMELA есть инструкция безусловного перехода

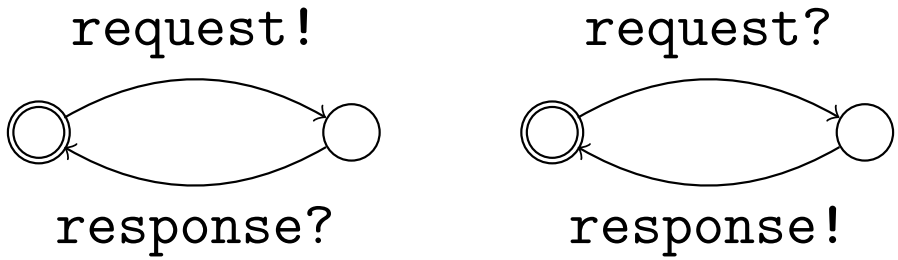
Эта инструкция работает точно так же, как и в C/C++:

```
...
LAB: a = b;
...
goto LAB;
...
```

В частности, если выполнение процесса — это зацикленное повторение какой-либо последовательности действий, то достаточно написать в теле процесса эту последовательность и в конце тела перейти в начало

Упражнение 3: передача сообщений

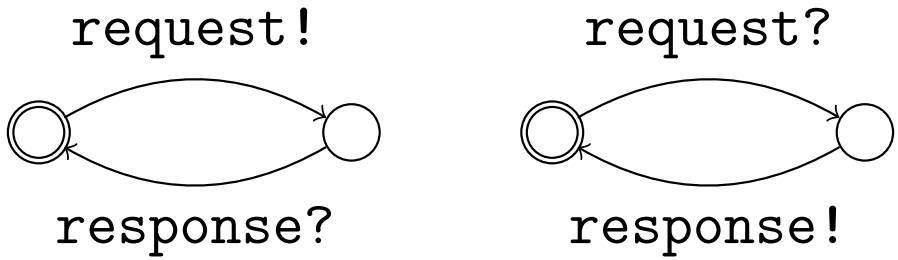
Система состоит из одного клиента, одного сервера и двунаправленного канала связи между ними
Клиент и сервер описываются левым и правым автоматами соответственно:



Запись `m!` означает, что при выполнении перехода сообщение `m` посылается в канал, запись `m?` — что сообщение `m` принимается из канала

Упражнение 3: передача сообщений

Система состоит из одного клиента, одного сервера и двунаправленного канала связи между ними
Клиент и сервер описываются левым и правым автоматами соответственно:



Выяснить,

- верно ли, что если клиент посылает сообщение `request`, то он обязательно примет сообщение `response`
- возможна ли ситуация, в которой и клиент, и сервер ожидают приёма сообщений

Рассмотреть два варианта устройства канала:

- синхронный
- асинхронный ёмкости 1

Упражнение 4: синхронные системы

По комнате летает два комара

В начальный момент времени оба комара не жужжат

Если комар не жужжит, в следующий момент времени он начинает жужжать

Если комар жужжит, в следующий момент времени он перестаёт жужжать

Жужжание комара описывается булевой переменной

Описать систему жужжания двух комаров с таким требованием: в каждый момент времени либо оба комара жужжат, либо оба комара не жужжат

Подсказка: в теле требований можно использовать метки состояний

Упражнение 5: переправа

Волк, коза, капуста и лодочник с лодкой стоят на левом берегу реки и хотят переправиться на правый

Только лодочник может грести

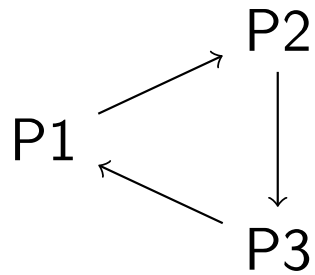
Лодка вмещает только двоих, включая лодочника

Нельзя оставлять волка с козой, а также козу с капустой на одном берегу без присмотра лодочника

Могут ли все переправиться на правый берег?

Упражнение 6: распределённые алгоритмы

Три процесса соединены друг с другом в кольцо однонаправленными асинхронными каналами связи ёмкости 1:

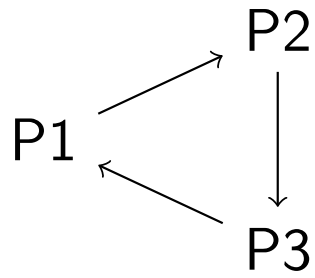


Каждый процесс имеет булеву переменную `b`, имеющую произвольное значение в начале работы системы, и **ровно два раза** делает следующее:

- посылает в канал значение `b`
- принимает из канала значение, и если принято `true`, то записывает `true` в переменную `b`

Упражнение 6: распределённые алгоритмы

Три процесса соединены друг с другом в кольцо однонаправленными асинхронными каналами связи ёмкости 1:



Убедиться, что

- каждый процесс рано или поздно выполнит всю свою последовательность действий
- (каждый процесс в конце работы хранит значение `true`) $\Leftrightarrow$ (хотя бы один процесс в начале работы хранил значение `true`)
- (каждый процесс в конце работы хранит значение `false`) $\Leftrightarrow$ (ни один процесс в начале работы не хранил значение `true`)

Ещё немного синтаксиса и семантики

Ранее на слайдах факт “процесс завершил работу” отождествлялся с фактом “процесс находится у метки состояния после всех его инструкций”

В общем случае всё работает сложнее: *в некоторых случаях* процесс по завершении работы полностью исчезает из системы, и тогда он не будет находиться у метки состояния после всех его инструкций

Если нет уверенности в том, удалился ли процесс, то следует выбрать другой способ выяснения того, завершился ли он (*например, использовать булевы переменные как флаги завершения*)