Введение в язык Promela и систему комплексной верификации Spin

И.В. Шошмина, Ю.Г. Карпов

Санкт-Петербургский государственный
политехнический университет

– 2009 –

Содержание

ПРЕДИСЛОВИЕ 4

1. Обзор языка спецификации моделей Promela 7

1.1. Графическая оболочка XSpin и модель “Hello world” *7*

1.2. Типы объектов языка Promela *10*

Типы процессов 10

Типы данных 12

Каналы 14

Взаимодействие рандеву 16

Правило выполнимости 17

Выражения 18

1.3 Составные операторы *20*

Блок `atomic` 20

Выбор по условию 21

Цикл 22

1.4 Некоторые примеры *23*

2 Описание верифицируемых свойств средствами Promela и SPIN 25

2.1 Оператор `assert` *26*

2.2 Метка заключительного состояния (`end`) *26*

2.3 Метки активного состояния (`progress`) *27*

2.4 Метка принимающего состояния (`accept`) *28*

2.5 Особый процесс `never` *28*

3 Примеры задач для верификации в Spin 33

3.1 Протокол выбора лидера в однонаправленном кольце *33*

Описание протокола 33

Описание модели протокола на языке Promela 34

Модель алгоритма выбора лидера на языке Promela 35

Пример симуляции модели выбора лидера 36

Верификация базовых свойств 38

Проверка LTL формул 38

3.2 Задача о фермере, волке, козе и капусте *43*

Описание задачи 43

Модель задачи о волке, козе и капусте на языке Promela 43

Поиск решения задачи средствами Spin 44

3.3 Криптографический алгоритм Нидхама-Шредера *47*

Описание алгоритма 47

Описание модели на языке Promela 48

Код алгоритма Нидхама-Шредера на языке Promela 51

Поиск криптографической атаки 54

3.4 Задача об обедающих философах *55*

Описание алгоритма 55

Описание модели на языке Promela 56
Обнаружение состояния общего голодания 58
3.5 Мир блоков *59*
Описание модели на языке Promela 60
Поиск решения задачи средствами Spin 62
Приложение 1. Параметры симуляции 63
Приложение 2. Параметры верификации базовых свойств 65
Литература 66

Предисловие

Появление на рынке многоядерных процессоров поставила широкие массы программистов перед проблемой корректного написания параллельных программ. Поэтому умение выражать свойства поведения параллельных систем и проверять выполнение этих свойств в разработанных программах становится необходимым, фактически, каждому программисту.

Данное пособие посвящено описанию программного средства верификации параллельных программ Spin. Пакет Spin позволяет:

- строить модели параллельных программ (протоколов, драйверов, систем логического контроля и управления) и широкого класса дискретных систем
- выразить требуемые свойства их поведения (так называемые “темпоральные свойства”) и
- автоматически (с помощью “push button technique” – “техники нажатия кнопки”) проверить выполнение темпоральных свойств параллельных систем на их моделях на основе формального подхода.

Spin – свободно распространяемый пакет программ для формальной верификации систем, разработанный в Исследовательском Центре Bell Labs Джерардом Холзманном (Jerard Holzmann), он широко применяется как в обучении студентов методам верификации, так и в промышленности для формального анализа свойств разрабатываемых протоколов и бортовых систем. Ежегодно проводятся конференции пользователей этого пакета, в литературе широко обсуждаются многочисленные примеры применения Spin для верификации сложных систем. Международная Ассоциация ACM (Association for Computing Machinery) наградила программное средство Spin престижной премией ACM Software System Award за 2001 г. В 1983 году этой премией была награждена ОС UNIX, в 1997 г. – Tcl/Tk, в 2002 г. – язык Java.

Литература по системе Spin на английском языке довольно обширна. Домашняя страница этой системы http://spinroot.com/. На этой странице можно найти ссылки на руководства по системе, ссылки на труды ежегодных конференций по этому продукту. Наиболее полным описанием системы Spin является монография Gerard J. Holzmann. The Spin Model checker. Авторам неизвестны руководства по системе Spin на русском языке.

Мы будем рассматривать Spin не с точки зрения его структуры и реализации, а с точки зрения его использования для верификации, проверки правильности программ. С помощью системы Spin выполняется проверка не самих программ, а их ***моделей***. Для построения модели по исходной параллельной программе или алгоритму пользователь (обычно вручную) строит представление этой программы на С-подобном входном языке пакета Spin, названном автором Promela (Protocol Meta Language). Это представление –программу на языке Promela можно считать моделью верифицируемой программы. Конструкции языка Promela просты, они имеют ясную и четкую семантику, позволяющую перевести (оттранслировать) любую программу на этом языке в систему переходов с конечным числом состояний, которая представляет собой модель переходов подлежащей верификации параллельной программы или алгоритма..

Аббревиатура Spin расшифровывается как Simple Promela Interpreter – *простой интерпретатор программ на языке Promela*.

Spin может использоваться в двух режимах – как симулятор и как верификатор. При симуляции Spin выводит информацию об одной конкретной траектории выполнения построенной модели - графическое представление поведения в виде Диаграмм Последовательностей Сообщений (Message Sequence Diagrams), возникающих при функционировании параллельных процессов по данной траектории. Графический интерфейс пользователя XSpin визуализирует запуски симуляций и упрощает процесс отладки модели.

Выполняя симуляцию и отладку систем, надо понимать, что никакое количество симуляций не может ***доказать*** свойства модели. Для доказательства свойств модели используется верификация – другой режим работы Spin. Верификатор пытается найти контрпример - неправильную, ошибочную траекторию поведения, опровергающую заданное пользователем свойство, анализируя все возможные поведения модели. Для этого он строит сложную конструкцию – синхронное произведение модели переходов анализируемой системы и автомата Бюхи, задающего все возможные некорректные поведения. В случае нахождения контрпримера симулятор Spin демонстрирует его пользователю в управляемом режиме симуляции. Таким образом, симуляция и верификация в Spin тесно связаны.

Разработанные на языке Promela модели отличаются от верифицируемых программ, обычно написанных на языках программирования высокого уровня, например, Java или C. Программы на Promela не имеют классов, они представляют плоскую структуру взаимодействующих параллельных процессов, имеют минимум управляющих конструкций. все переменные имеют конечные области определения. Поэтому такую программу можно считать моделью анализируемой системы они представляют абстракцию системы, в которой пользователь должен отразить те аспекты и характеристики реальной проверяемой системы, которые значимы для заданных для верификации свойств. Построенная модель может содержать траектории, которые не относятся к реализации проверяемой системы в исходном языке программирования. Например, такие траектории могут включать явную спецификацию возможных ситуаций при наихудшем поведении среды, в модели может явно или неявно указываться спецификация свойств справедливости.

Описание проверяемой системы на языке Promela должно сохранять существенные свойства этой системы. Можно сказать, что качество результата проверки моделей программ, представленных на Promela, полностью зависит от степени адекватности построенной модели.

Таким образом, модель программной системы, верифицируемая с помощью системы Spin, и реальная реализация системы принципиально отличаются. Верифицируемая модель может строиться на этапе начальной разработки структуры системы (при создании прототипа). Окончательная же система, как правило, отличается существенной детализацией: произвольными объемами буферов, наличием вещественных переменных, более богатым спектром управляющих конструкций, динамическим порождением произвольного числа процессов и т.п.. Достаточно трудно предложить средство автоматического перехода от реализации системы к ее модели в виде программы на языке Promela. Формальных правил “правильного” построения модели по исходной программе нет, и не может быть, этому можно научиться только на примерах. Одна из целей пособия – дать такие примеры. Построение модели сложной системы на языке Promela, сохраняющей нетривиальные свойства этой системы, является искусством, которому нужно учиться.

Отметим, однако, что в последнее время многие крупные разработчики критического ПО (например, бортовых систем управления для космических и летательных аппаратов), в частности, фирма Rockwell Collins, используют новую технологию разработки ПО - MDD (Model Driven Development). В соответствии с этой технологией формальная модель системы является первичной: сначала именно формальная модель строится для будущей системы и верифицируется – на ней проверяются все требуемые свойства, и затем эта же модель является основой для генерации “ядра” реализации, к которому могут быть добавлены дополнительные функции таким образом, чтобы эти функции не нарушали проверенных свойств.

Структура пособия

Пособие состоит из трех частей. В первой части приводится обзор основных средств языка Promela. Вторая часть пособия на нетривиальных примерах демонстрирует разработку нескольких моделей. В частности, строятся модели трех программных систем: протокола

выбора лидера, известной старинной логической задачи “фермер, волк, коза и капуста” и криптографического протокола аутентификации Нидхэама-Шредера. Первая система верифицируется, т.е. показывается, как с помощью системы Spin проверяются те свойства, которые должны соблюдаться в протоколе. Для второй системы верификатор автоматически находит нетривиальное решение логической задачи, в третьем примере демонстрируется, как верификатор автоматически находит атаку на криптографический протокол аутентификации, т.е. такое поведение злоумышленника, которое нарушает конфиденциальность криптографического протокола. Стоит отметить, что некорректность этого протокола (т.е. возможная стратегия атаки) не были известны в течение более 10 лет. Кроме этих моделей приводятся и комментируются другие интересные модели, на которых можно научиться использованию системы Spin для верификации ПО различного назначения.

Последняя часть пособия – это Приложение. Здесь объясняется, как установить систему Spin на компьютере студента. Система Spin является бесплатно распространяемой и не требует лицензирования.

Изучать пособие нужно, работая на компьютере. Установив систему Spin, студенту следует строить примеры, описанные в пособии. Все приведенные здесь программы – рабочие, их нужно ввести в поле редактора, запустить и при их анализе пытаться получить те же результаты, которые приведены в тексте.

Данное пособие разработано при частичной поддержке фирмы Интел по Договору о выполнении работы по модернизации лекционного курса «Верификация параллельных программных и аппаратных систем», читаемого на факультете Технической кибернетики СПбГПУ для магистров по направлению 552822 — *Распределенные автоматизированные системы*, (Договор № SPB/R&D/139/ 2006 от 16.11.2006 между Санкт Петербургского государственного политехнического университета и Интел Текнолоджис Инк.).

1. Обзор языка спецификации моделей Promela

Язык Promela - абстрактный язык спецификации алгоритмов. Абстрагирование направлено на то, чтобы с помощью минимальных выразительных средств (параллельные процессы и коммуникация с помощью каналов, простейшие типы данных с конечным числом возможных значений и простые управляющие структуры) строить такие абстрактные модели реальных параллельных и распределенных систем, которые легко представляются формальной моделью с конечным числом состояний (структурой Крипке и автоматами Бюхи). Таким образом, Promela является не языком реализации систем (языком программирования), но *языком описания моделей распределенных систем*.

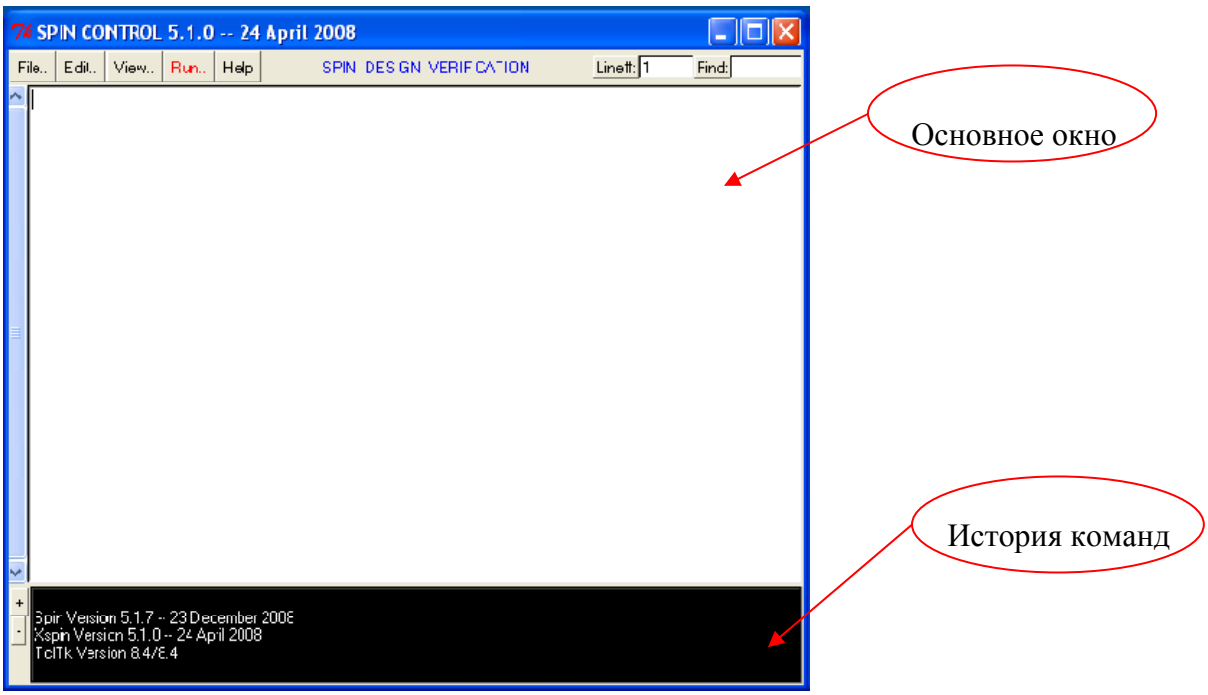
Язык Promela включает примитивы для создания процессов и богатый набор примитивов для межпроцессного взаимодействия. Базовые блоки моделей в Promela составляют асинхронные процессы, каналы сообщений, синхронизирующие утверждения, структурированные данные. В языке также поддерживается спецификация структур недетерминированного управления.

Таким образом, Promela содержит некоторые средства, которые отсутствуют в обычных, широко распространенных языках программирования. Эти средства способствуют конструированию моделей распределенных систем с высоким уровнем абстракции. В то же время в языке отсутствует множество возможностей, которые есть в большинстве языков программирования, например, функции, возвращающие значения, указатели на данные и функции, сложные структуры данных и т. п. В язык намеренно не включено понятие времени или часов (свойства реального времени не проверяются в системе Spin); отсутствуют операции с плавающей точкой (чтобы ограничиться моделями с конечным числом состояний); ограничено количество вычислительных функций. Подобные ограничения делают относительно сложной реализацию в языке Promela, например, вычисления квадратного корня, но относительно простой реализацию, напритмер, клиент-серверного взаимодействия в сети. "Перекос" в сторону средств взаимодействия процессов в языке объясняется направленностью языка Promela на верификацию именно параллельных и распределенных систем, в первую очередь синхронизацию и координацию параллельно протекающих процессов.

На практике именно разработка корректной структуры синхронизации и координации для программного обеспечения параллельных и распределенных систем оказывается гораздо более сложной и чреватой ошибками, чем разработка последовательных вычислений. Логическая верификация взаимодействия в параллельных системах, несмотря на значительную вычислительную сложность, может быть сегодня выполнена на основе новой техники верификации “model checking” более надежно и тщательно по сравнению с верификацией последоватедбных вычислительных процедур. Именно эту технику Spin использует при верификации. Отметим, например, что большинство ошибок, выявленных с помощью Spin при верификации бортовой системы управления космического аппарата Deep Space 1, были ошибками, допущенными разработчиками этой системы в силу непредвиденных взаимодействий и перекрытий выполнений параллельных процессов.

1.1. Графическая оболочка XSpin и модель “Hello world”

Программное средство Spin можно использовать в командной строке, указывая различные ключи. Однако мы рекомендуем воспользоваться графической оболочкой XSpin. Запустите XSpin (установка и запуск программы XSpin описаны в Приложении 1 “Установка и настройка системы Spin с GUI XSpin”).



Основное окно XSpin является простым текстовым редактором. В меню *Edit* доступны обычные опции для работы с текстом: *Copy*, *Cut*, *Paste*. С помощью меню *View* можно изменять размеры шрифта в основном окне. Под основным окном находится окно с историей команд (`command-log`). В дальнейшем все выполняемые команды будут отображаться в окне команд.

Рассмотрим пример простейшей программы на языке Promela. Наиболее часто встречающимся первым примером в руководствах по языкам программирования является программа, которая печатает строку hello world на терминал пользователя. Традиция началась с руководства для языка программирования C (1978), авторами которого были Керниган и Ритчи. С тех пор этот пример кочует по руководствам языков программирования. Специфицируем эту маленькую программу как модель на языке Promela:

```
active proctype example()
{
      printf("MSC: hello world\n")
}
```

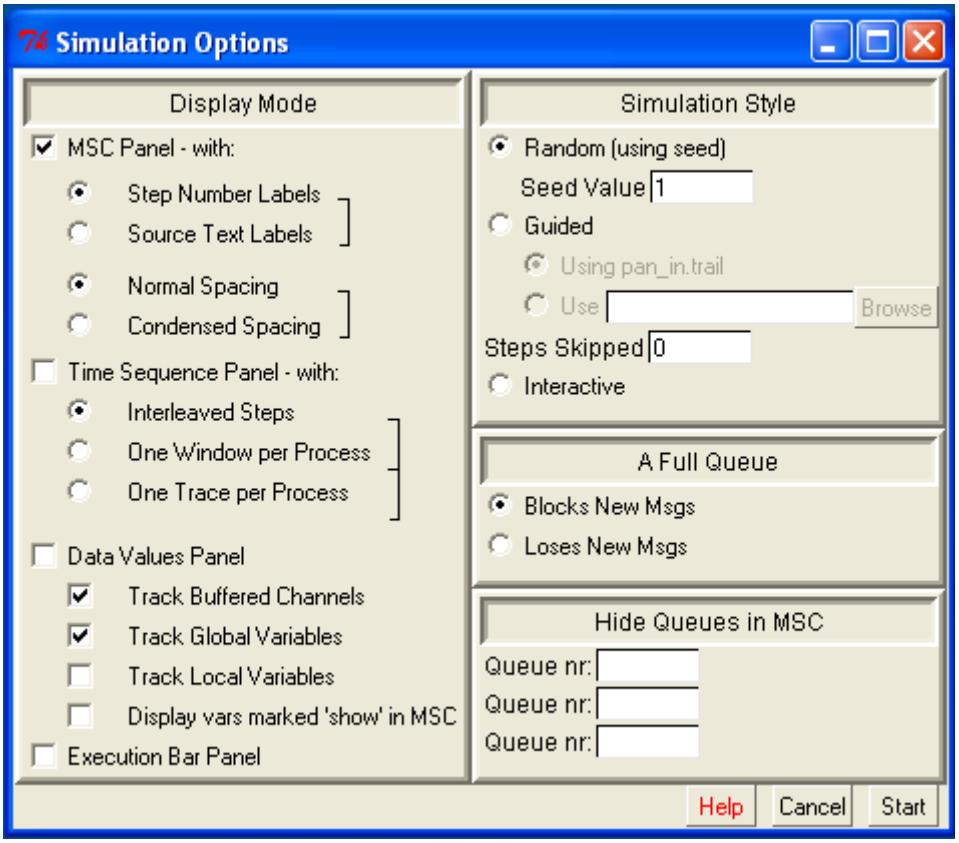
Модели на Promela используются для описания поведения систем потенциально взаимодействующих процессов, т.е. нескольких асинхронно выполняемых потоков. Поэтому первичная единица выполнения в Spin – это процесс. В отличие от C, здесь нет основной процедуры `main`. Служебное слово `proctype` описывает в этом примере новый тип процессов с названием `example`. Префикс `active` говорит о том, что один экземпляр процесса типа `example` должен быть создан сразу в начале работы модели. Если префикс опущен, то экземпляр процесса не будет создан при таком объявлении. Создание экземпляров процессов, объявленных с помощью служебного слова `proctype`, может быть выполнено другими средствами.

Введем текст модели в основное окно XSpin. Сохраним модель в файле `hello` при помощи меню *File*. Можно воспользоваться и другим текстовым редактором, обновляя файл с помощью опции *ReOpen* в меню *File* XSpin.

Для того чтобы проверить синтаксические ошибки в файле с моделью на Promela, выберите в меню *Run* пункт *Run Syntax Check*. В нашем файле `hello` синтаксических ошибок нет, и в командном окне выведется сообщение “`no syntax errors`”.

Spin может продемонстрировать один из возможных вариантов поведения модели – такой режим работы Spin называется *симуляцией* (пункт *(Re)Run Simulation* меню *Run*).

До запуска симуляции необходимо проверить параметры этого режима (пункт *Set Simulation Parameters* меню *Run*).



В окне параметров симуляции на панели режима отображения (*Display Mode)* задаются окна, которые необходимо выводить при симуляции – панель диаграммы взаимодействия *MSC Panel,* режим вывода событий отдельных процессов *Time Sequence Panel,* панель для просмотра значений данных *Data Values Panel* и панель выполненных шагов модели *Execution Bar Panel* (о параметрах симуляции см. Приложение 1).

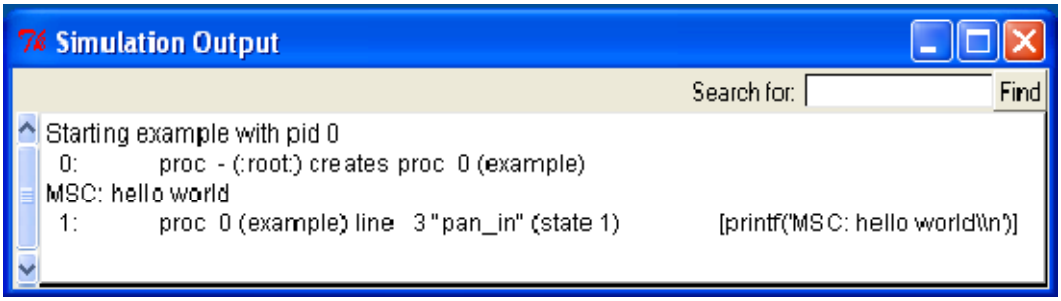
В нашей первой модели присутствует вывод на панель диаграммы взаимодействия, о чем свидетельствует синтаксис оператора `printf`

```
printf("MSC: ")
```

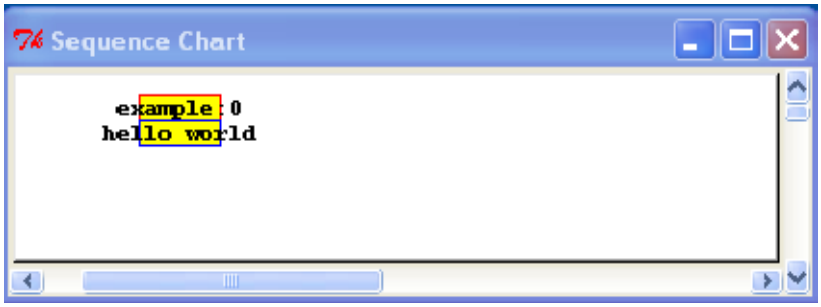
поэтому необходимо установить флаг *MSC Panel*.

На панели *Simulation Style* устанавливается тип симуляции. Возможны три вида симуляции: случайная *Random* – все недетерминированные решения определяются случайным образом, управляемая *Guided*, интерактивная *Interactive* – все недетерминированные решения запрашиваются у пользователя в интерактивном режиме. Установим случайный тип симуляции *Random simulation*. Запустим модель на симуляцию, нажав кнопку *Start*.

При симуляции Spin найдет некоторое возможное вычисление модели. В окне вывода симуляции (*Simulation Output*) отображаются все события модели, произошедшие во время этого конкретного вычисления. Нажмите *Run*, запустив тем самым режим симуляции. В данном конкретном примере на шаге 1 был выполнен оператор `printf`.



Вывод в главном окне симуляции является исчерпывающим, в том смысле, что по нему можно точно установить всю последовательность событий конкретного вычисления. Однако формат представления не удобен для пользователя.



Второе окно, открывшееся при запуске панель диаграммы взаимодействия, *Sequence Chart*, наглядно демонстрирует результат симуляции нашего первого примера. Единственный процесс типа `example` с идентификатором `0` вывел сообщение `hello world`.

1.2. Типы объектов языка Promela

Программа, написанная на Promela, состоит из трех основных типов объектов: *процессы* (*processes*), *каналы сообщений* (*message channels*) и *переменные* (*data objects*). Процессы задают поведение, каналы и глобальные переменные определяют окружение, в котором выполняются процессы.

Типы процессов

Типы процессов описываются при помощи объявления `proctype`. Для разумного использования модели на Promela должно быть хотя бы одно объявление типа процесса и хотя бы один экземпляр процесса какого-нибудь типа. Процессы всегда объявляются глобально.

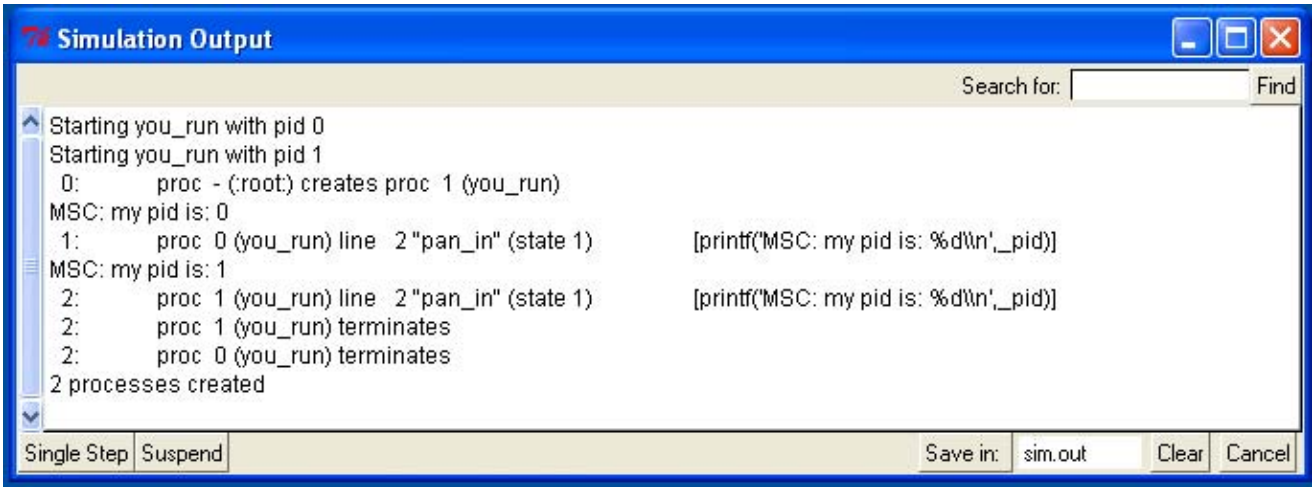
Существует несколько способов создания экземпляров типа процесса. В приведенном ниже примере создается 2 экземпляра типа процесса `you_run` с помощью префикса `active`:

```
active [2] proctype you_run(){
        printf("MSC: my pid is: %d\n", _pid)
}
```

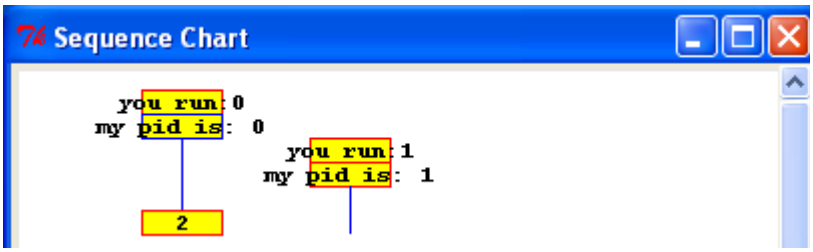
Каждый запущенный процесс имеет уникальный идентификатор, значение которого неотрицательно. Идентификатор процесса хранится в зарезервированной локальной переменной `_pid`.

Тело процесса может состоять из *объявлений данных* и операторов, оно может быть и пустым. Состояние переменной или канала сообщений изменяется или проверяется только в процессах. Точка с запятой является *разделителем операторов* в теле процесса (но не указателем конца оператора, поэтому после последнего оператора точка с запятой может отсутствовать). Лишние точки с запятой не являются ошибкой, поскольку в Promela допускается пустой оператор.

В программах на Promela существует два различных разделителя операторов: стрелка “->” и точка с запятой “;”. Эти два разделителя эквивалентны. Стрелка иногда используется как неформальный способ указания на причинно-следственное отношение между двумя операторами: если предыдущий оператор выполняется, то управление передается на выполнение второго оператора (что, собственно, соответствует и семантике точки с запятой).



В главном окне симуляции отобразится информация о том, что было создано два процесса типа `you_run`.

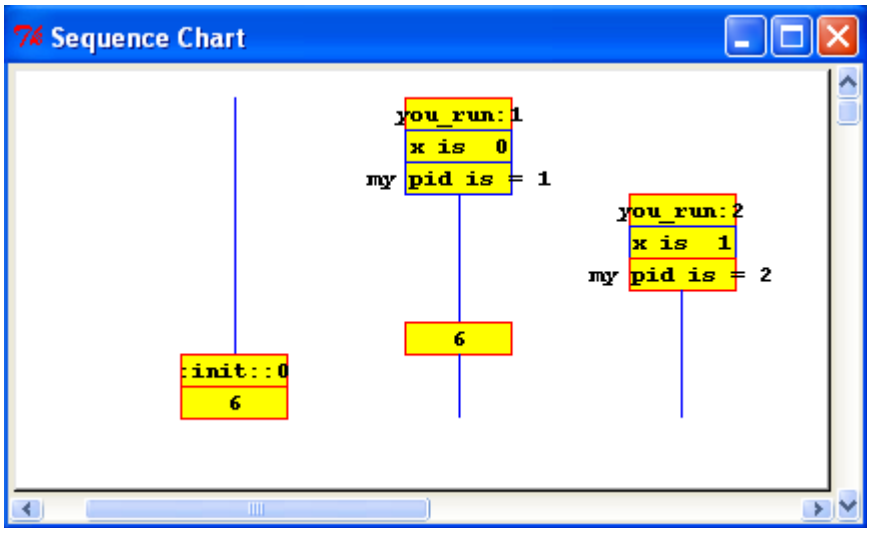


На шаге 1 процесс с идентификатором 0 выполнил оператор `printf`, на шаге два процесс с идентификатором 1 выполнил оператор `printf`, после чего оба процесса завершили работу. Поскольку процессы асинхронны, то очевидно, что последовательность выполнения операторов `printf` могла бы быть противоположной. На панели диаграммы взаимодействий (Sequence Chart) каждый столбец отображает один запущенный процесс.

Процесс также можно запустить, воспользовавшись оператором `run`.

```
proctype you_run(byte x)
{
       printf("MSC: x is  %d\n",  x);
       printf("MSC: my pid is = %d\n", _pid)
}
init {
        run you_run(0);
        run you_run(1)
}
```

В этом примере процесс `init` запускает два процесса типа `you_run` и передает каждому из них входной параметр. Все запущенные процессы работают параллельно. Процесс `init` – базовый процесс в Promela, который всегда активируется в начальном состоянии модели. Процессу `init` нельзя передать параметры или создать его копию. Если он есть в модели, то его идентификатор всегда равен 0.



Данное решение имеет недостаток, связанный с созданием лишнего процесса (процесса `init`), что увеличивает размер модели. Размер модели не влияет на режим симуляции, но отражается на времени верификации.

Операторы `run` используются в любых процессах для того, чтобы породить новый экземпляр процесса заданного типа, т.е. он может встретиться не только в процессе `init`. Выполняющийся процесс завершается, когда достигает конца тела описания своего типа процесса, но не позже своего процесса-родителя. Количество процессов, которое может быть создано в Spin ограничено числом 255.

Типы данных

Типы данных Promela близки к данным языка C (см. табл. 1) .

Табл. 1. Типы данных и диапазоны значений для компьютера с длиной слова 32 бит.

Тип данных	Диапазон	Пояснение
`bit`	0,1	
`bool`	false,true	
`byte`	0..255	
`chan`	1..255	Число каналов
`mtype`	1..255	Число значений перечислимого типа
`pid`	0..255	Число возможных процессов
`short`	-2^{15} .. $2^{15}-1$	
`int`	-2^{31} .. $2^{31}-1$	
`unsigned`	0 .. $2^{32}-1$	

В Promela существует два уровня доступа к объектам: локальный и глобальный. Объект, объявленный глобально, будет доступен всем процессам.

Используются ли *переменные* для хранения глобальной информации о системе в целом, или информации, касающейся конкретного процесса, зависит от того, где помещается *объявление переменной*. Переменная является *глобальной*, если она объявлена вне описания процесса, и

локальной, если она объявлена в описании процесса. Не существует возможности ограничить доступ к локальной переменной лишь для части модуля, описывающего процесс (т.е. не существует аналога блока или области видимости). Все переменные инициализируются нулями (для булевой переменной это *false*).

```
bool flag;        /* Примеры объявлений переменных */
int state;
byte msg;
```

В языке допускаются комментарии только такого вида: /* комментарий */.

В Promela поддерживаются только одномерные массивы например,

```
byte state[N]
```

объявляет массив с именем `state` из `N` байт, к которому можно обращаться следующим образом

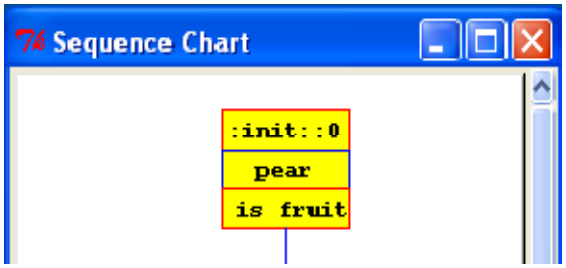
```
state[0] = state[3] + 5 * state[3*2/n]
```

где `n` является константой или переменной. Индексом массива может являться любое выражение, которое вычисляет целочисленное значение. При значении индекса вне диапазона `0.. N-1` результат не определен; наиболее вероятно, что это вызовет ошибку при выполнении программы (runtime error).

Многомерные массивы могут быть заданы неявно с помощью конструкции `typedef` (см. пример алгоритма Нидхама-Шредера п.3.3).

Перечислимый тип `mtype` (mnemonic type, мнемонический тип) может содержать символьные значения переменных, которые задаются при помощи одного или нескольких объявлений. Фактически, это аналог перечислимого типа в других языках программирования. Все переменные, объявленные с помощью `mtype`, представляют собой одно множество перечислимого типа, независимо от числа объявлений. При помощи всех объявлений типа `mtype` можно задать не более 255 значений:

```
mtype = { apple, pear, orange, banana };
mtype = { fruit, vegetables, cardboard };

init {
        mtype n = pear; /* инициализация переменной n значением pear */
        printf("MSC: %e ", n);
        n = fruit;      /* присвоение переменной n значения fruit */
        printf("MSC: is %e\n ", n)
}
```



Синтаксис оператора `printf` подобен аналогичному оператору в C. У этого оператора два аргумента: строка и список аргументов. В строке задаются форматы выводимых переменных: `d` – целое в десятичном формате, `i` – беззнаковое целое значение, `c` – один символ, `e` – константа типа `mtype`. Как уже отмечалось ранее для вывода сообщения на диаграмме взаимодействия XSpin строка в `printf` должна начинаться с пяти символов “`MSC: `”.

Каналы

Каналы сообщений используются для моделирования передачи данных от одного процесса к другому. Они объявляются локально или глобально при помощи служебного слова `chan`:

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

Здесь объявлен канал `qname` с буфером 16, т.е. этот канал может хранить до 16 сообщений типа `short`, т.е. в канале может находиться не более 16 непрочитанных сообщений .

Для работы с каналами существуют операторы посылки и получения сообщений. Оператор "`!`":

```
qname ! expr
```

посылает сообщение со значением переменной `expr` в только что объявленный канал `qname`, добавляя это значение к концу очереди в канале. По умолчанию оператор выполняется, если канал назначения не переполнен, в противном случае он блокируется. Каналы передают сообщения в порядке FIFO: первым вошел – первым вышел.

Оператор приема сообщения "`?`":

```
qname ? msg
```

принимает сообщение из начала буфера канала и сохраняет его в переменной `msg`. Выполнение опрератора приема сообщения возможно только в случае, если канал не пуст.

Для передачи составных сообщений, т.е. сообщений, содержащих конечное число полей, используются каналы, объявленные подобно данному:

```
chan pname = [16] of { byte, int, chan }
```

В примере объявляется канал `pname` для сообщений, каждое из которых содержит три поля – одно восьмибитное значение (типа `byte`, стоящее в начале сообщения), одно 32-битное значение (типа `int`) и имя канала. Передача идентификатора (имени) канала от одного процесса другому возможна посредством сообщения (как в данном примере) или в качестве параметра экземпляра процесса (см. пример алгоритма выбора лидера – п.3.1). Заметим, что использование массивов как поля сообщения в явном виде не допускается.

При посылке нескольких значений в одном сообщении задается список переменных через запятую

```
pname ! expr1,expr2,expr3
```

В другом процессе можно принять это сообщение, присвоив три переданные значения трем различным переменным:

```
pname ? var1, var2, var3
```

Часто первое поле сообщения используется для задания *типа сообщения* (фактически, для связывания с сообщением некоторой характеристики). Эти данные объявляются при помощи типа `mtype`:

```
/* объявляется тип сообщения: */
mtype = { ack, nak, err, next, accept }
/* объявляются две переменные этого типа: */
mtype msgtype1, msgtype2;
```

При использовании типа передаваемых данных `mtype` в объявлении канала соответствующие поля сообщений будут всегда интерпретированы символически, а не численно. Например:

```
chan tname = [4] of { mtype, int, bit };
```

Здесь по каналу `qname` будут пересылаться сообщения, состоящие из трехполей: первое – типа `mtype`, второе – целое значение, третье – битовое значение. Для задания операций посылки и получения сообщения определенного типа используется следующее обозначение

```
tname ! msgtype (data, b)
```

Эта запись эквивалентна следующей альтернативной записи:

```
tname ! msgtype, data, b
```

Некоторые аргументы операторов как посылки, так и приема сообщений могут быть константами:

```
tname ! ack, var, 0    /* По каналу tname передается сообщение
                          из двух констант (ack типа mtype и 0) и
                          значения целой переменной var */

tname ? ack (data,1)   /* Из канала qname читается сообщение, в котором
                          первым элементом должна быть константа ack типа
                          mtype, вторым – произвольное значение типа int,
                          которое запишется в переменную data, третьей частью
                          принятого сообщения должна быть битовая константа 1
                          */
```

Оператор получения сообщения при наличии констант будет выполним, только если сообщение в начале буфера канала в соответствующих полях имеет значения заданных констант. В противном случае он будет блокирован. Например, приведенный выше оператор получения сообщения будет невыполним, если в начале буфера находится сообщение `<ack,15,0 >`.

Если оператор окажется невыполнимым, то процесс, пытающийся его выполнить, будет приостановлен, пока оператор не станет выполнимым. Таким образом, можно моделировать коммуникацию «точка-точка» нескольких процессов, связанных одним каналом.

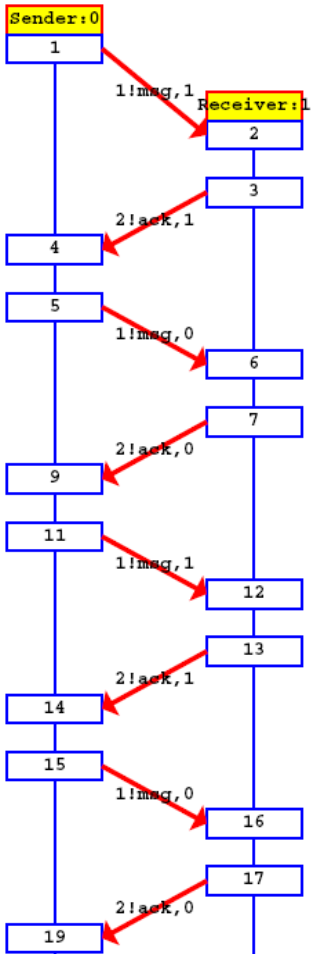
Рассмотрим упрощенную реализацию протокола альтернирующего бита (Bartlett, Scantlebury и Wilkinson [1]):

```
mtype = { msg, ack };          /*объявляем два возможных типа сообщения */

chan    to_sndr = [2] of { mtype, bit };   /*объявляем канал отправителя */
chan    to_rcvr = [2] of { mtype, bit };   /*объявляем канал получателя */

active proctype Sender(){      /* процесс отправителя */
again:  to_rcvr!msg,1;         /* отправляем сообщение, помеченное битом 1 */
        to_sndr?ack,1;         /* получаем подтверждение, помеченное битом 1 */
```

```
        to_rcvr!msg,0;        /* отправляем сообщение, помеченное битом 0 */
        to_sndr?ack,0;        /* получаем подтверждение, помеченное битом 0 */
        goto again
}

active proctype Receiver(){  /* процесс получателя */
again:  to_rcvr?msg,1;        /* получаем сообщение, помеченное битом 1 */
        to_sndr!ack,1;        /* отправляем подтверждение, помеченное битом 1 */
        to_rcvr?msg,0;        /* получаем сообщение, помеченное битом 0 */
        to_sndr!ack,0;        /* отправляем подтверждение, помеченное битом 0 */
        goto again
}
```

В алгоритме альтернирующего бита процесс-отправитель попеременно отправляет сообщения, помеченные то битом 1, то битом 0, и ожидает соответствующие подтверждения. Процесс-получатель получает сообщения, помеченные то битом 1, то битом 0, и отсылает процессу-отправителю на них подтверждения. В полной версии протокола сообщения могут быть потеряны, но здесь потери канала не моделируются. Бесконечный цикл реализован в этой модели при помощи оператора перехода `goto` и локальной метки `again` в каждом процессе.

В языке Promela для каналов определено несколько функций (`len`, `empty`, `nempty`, `full`, `nfull`). Например, предопределенная функция `len(qname)` возвращает количество сообщений, хранящихся в канале `qname` в текущий момент времени. Если `len` используется как оператор по правую сторону присваивания, то он будет невыполимым, если канал пуст, т.к. он возвращает нулевой результат, который по определению означает, что оператор временно невыполним. Следующий пример показывает, как отправить сообщение `msgtype` в случае, если канал `qname` не переполнен:




```
(len(qname) < MAX) -> qname ! msgtype
```

Здесь если доступ к каналу qname разделяется несколькими процессами, то выполнение второго оператора необязательно будет происходить сразу после выполнения первого оператора проверки. Например, это случится, если другой процесс пошлет сообщение в канал `qname` (и число сообщений в нем станет равно `MAX`) сразу после того, как данный процесс определит, что канал не полон.

Взаимодействие рандеву

Ранее мы рассматривали только асинхронные взаимодействия между процессами через канал сообщений, декларируемый как

```
chan qname = [N] of { byte }
```

где `N` является положительной константой, которая определяет размер буфера. Установив размер буфера, равным 0:

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

определим *рандеву-канал* (который в данном примере может передавать однобайтное сообщения). Поскольку буфер рандеву-канала равен нулю, то такой канал может передавать, но не может

хранить сообщения. Взаимодействия процессов по рандеву-каналам по определению синхронны. Рассмотрим следующий пример.

```
#define msgtype 1
chan name = [0] of { byte, byte };
proctype A()
{     name ! msgtype(4);
      name ! msgtype(1)
}
proctype B()
{     byte state;      name ? msgtype(state)
}
init
{     atomic { run A(); run B() }
}
```

Канал `name` объявлен здесь как глобальный рандеву-канал. Два процесса синхронно выполнят свои первые операторы: подтверждение связи («рукопожатие») по сообщению `msgtype` и передачу значения 4 в локальную переменную `state`. Второй оператор посылки в процессе `A` невыполним, потому что отсутствует соответствующая операция приема сообщения в процессе `B`.

Если бы канал `name` был определен с ненулевым размером буфера, поведение было бы отличным. Допустим, что размер буфера равен 2, тогда процесс типа `A` может закончить свое выполнение даже до того, как процесс B начнет работу. Если назначить размер буфера, равным 1, то последовательность событий будет следующей. Процесс типа `A` может закончить свое первое действие посылки, но он блокируется на втором действии, потому что теперь канал полон. Процесс типа `B` читает первое сообщение и завершается. В этой точке `A` становится опять выполнимым и завершается, оставляя свое последнее сообщение в канале.

Взаимодействия рандеву бинарные: только два процесса, отправитель и получатель, могут быть синхронизированы при рандеву «рукопожатии». Ниже будет рассмотрен пример способа использования рандеву-канала для построения семафора –п.1.3.

Правило выполнимости

Promela основывается на семантике *выполнимости*, которая обеспечивает базовые средства в языке для моделирования синхронизации процессов. В зависимости от состояния системы, любой оператор в модели Spin или *выполним*, или *блокирован*. Определены четыре основных типа *операторов* в Promela: оператор вывода переменных `printf`, оператор присваивания, операторы ввода/вывода (при передаче данных по каналам) и операторы-выражения. Если процесс достиг точки кода, где содержится невыполнимый оператор, то процесс *блокируется*. Он может стать снова выполнимым, если ДРУГОЙ активный процесс выполнит действия, позволяющие оператору, блокированному в данном процессе, выполняться далее.

Благодаря правилу выполнимости, запись многих операций в Promela упрощается. Например, вместо того, чтобы писать цикл активного ожидания:

```
while (a != b) ->skip
                    /* ожидать до тех пор, пока не выполнится условие a==b */
```

где `skip` является пустым (нулевым) оператором, в Promela достаточно написать выражение:

```
(a == b)
```

Это пример “пассивного” ожидания, когда процесс приостанавливается до выполнения условия (в данном случае - равенства a и b).

Рассмотрим следующую модель, в которой два процесса разделяют доступ к глобальной переменной `state`.

```
byte state = 1;

active proctype A(){
     byte tmp;
      (state==1) -> tmp = state; tmp = tmp+1; state = tmp
}

active proctype B(){
      byte tmp;
      (state==1) -> tmp = state; tmp = tmp-1; state = tmp
}
```

Процессы ожидают выполнения условия `(state == 1)`. Если каждый из процессов пройдет это условие до того, как другой изменит значение `state`, выполнив оператор `state = tmp`, то оба процесса завершатся, хотя итоговое значение глобальной переменной `state` будет непредсказуемо. В результате работы этой программы после ее завершения переменная `state` может иметь любое значение: 0, 1 или 2. Если же один из процессов успеет изменить значение переменной `state` до проверки условия другим процессом, то другой процесс заблокируется.

Выражения

Для построения выражений в языке Promela используются операторы, схожие с операторами языка C. В порядке приоритетов они приведены в таблице 2. Как обычно, в случае сомнений о порядке выполнения операторов и для большей ясности рекомендуется использовать круглые скобки.

В ыражения в Promela рассматриваются как утверждения, т.е. проверяются на истинность-ложность в любом контексте. Выражение выполнимо тогда и только тогда, когда его булево значение истинно (true), что эквивалентно любому ненулевому значению целочисленной переменной.

Табл. 2. Допустимые операторы предшествования в выражениях на языке Promela (по старшинству от старших к младшим).

Операторы	Направление действия	Комментарий
`() []`	слева направо	скобки, скобки массивов
`! ++ --`	справа налево	отрицание, увеличение на 1, уменьшение на 1
`* / %`	слева направо	умножение деление модуль
`+ -`	слева направо	прибавить, отнять
`<< >>`	слева направо	сдвиг влево, сдвиг вправо
`< <= > >=`	слева направо	операторы отношений
`== !=`	слева направо	равенство, неравенство
`&`	слева направо	побитовое И
`^`	слева направо	побитовое исключающее ИЛИ
`|`	слева направо	побитовое ИЛИ
`&&`	слева направо	логическое И

`||`	слева направо	логическое ИЛИ
`-> :`	справа налево	операторы условий
`=`	справа налево	присваивание

Оператор *присваивания*

```
variable = expression
```

не рассматривается как выражение, так же, как и оператор печати. Эти операторы всегда выполнимы. При выполнении оператора присваивания в Promela сначала оценивается выражение, стоящее справа (сначала инициализируются значения переменных выражения, если это не было сделано ранее, потом к ним применяются соответствующие операторы из табл. 2). После оценки полученный результат выражения, в случае необходимости, обрезается до значения того типа, которому принадлежит переменная `variable`, и переменная `variable` получает это значение.

Использование инкремента и декремента в Promela отличается от их использования в C: они могут быть только постфиксными (т.е. можно записать a++, но нельзя ++a) и могут использоваться только в выражении, но не в операторе присваивания.

*Входные/*выходные *операторы* выполняются только в случае наличия возможности отправки и получения сообщения, иначе процесс, выполняющий соответствующий оператор, блокируется (см. п.1.2, каналы). В некоторых случаях требуется протестировать возможность операции отправки или получения, не выполняя ее. Для этого входные/выходные операторы, фактически, трансформируются в обычные выражения взятием аргумента оператора в квадратные скобки. Таким образом, при использовании квадратных скобок во входных/выходных операторах сам оператор не выполняется, а лишь вычисляется его значение как выражения, например:

```
qname ? [ack, var, 0]
```

Смысл этого логического выражения состоит в проверке того, что очередным сообщением в канале с именем `qname` является структура, состоящая из мнемонического значения `ack`, некоторого значения переменной `var` и значения 0. Если утверждение выполнимо, то возвращается единица, иначе возвращается нуль.

Похожие на это условные выражения C-подобного вида

```
(qname ? var == 0)          /* синтаксическая ошибка */
```

или

```
(a > b && qname ! 123)      /* синтаксическая ошибка */
```

в Promela недопустимы, поскольку они не могут быть вычислены без побочных эффектов (попытки выполнить операции ввода/вывода вместо того, чтобы просто проверить, что очередное сообщение содержит 0 в первом случае или может быть послано – во втором). Кроме того, сами операторы отправки и получения данных не являются выражениями.

Для *вывода значений переменных или текста* используется оператор `printf`. По своему действию на состояние системы `printf` подобен пустому оператору `skip`. Оба этих оператора используются в моделях, когда необходимо осуществить всегда выполнимый шаг. Вывод текста `printf` производится только в режиме симуляции Spin, поскольку является побочным действием оператора.

1.3 Составные операторы

В части 1.2 были рассмотрены базовые операторы, существующие в Promela. Кроме них в языке определены составные операторы: последовательности вида `atomic`, детерминированные шаги, структура выбора и структура повторения.

Блок `atomic`

Последовательность операторов, заключенных в фигурные скобки, перед которой стоит ключевое слово `atomic`,, представляет блок кода, который должен выполняться как один неделимый модуль, не чередующийся с другими процессами. Действие `atomic` подобно использованию семафора. Выполнение последовательности операторов внутри `atomic` является неделимым “с точки зрения” других процессов: другие процессы “видят” разделяемые глобальные переменные и каналы, используемые в этом блоке, либо до, либо после выполнения всей последовательности операторов внутри блока `atomic`. В случае если какой-либо оператор внутри `atomic` не является выполнимым, вся последовательность операторов не выполняется. Модифицируем пример с разделяемым использованием общей для двух процессов переменной `state`.

```
byte state = 1;

active proctype A(){
      atomic {
        (state==1) -> state = state+1
      }
}

active proctype B(){
      atomic {
        (state==1) -> state = state-1
      }
}
```

Использование блока `atomic` позволяет здесь предотвратить доступ конкурирующего процесса к глобальной переменной. При запуске этой модели может быть недетерминировано выбран либо процесс `A`, либо процесс `B` для того, чтобы выполнить все операторы блока `atomic`. Если начал выполняться процесс `A`, то он завершится, после чего запустится процесс `B`, который, однако, будет приостановлен на проверке условия `(state == 1)`. Аналогично, если начал выполняться процесс `B`, то он завершится, после чего процесс `A` будет приостановлен на проверке такого же условия. Итоговое значение переменной `state` будет либо 0, либо 2, в зависимости от того, какой из двух процессов выполнится.

Блоки `atomic` являются важным инструментом понижения сложности моделей верификации – уменьшения числа глобальных состояний строящейся Spin’ом формальной модели переходов: блоки `atomic` ограничивают количество чередований (интерливинга).

Однако, надо понимать, что интерливинг реально возникает в распределенных системах и порождает неопределенность в выполнении параллельных процессов, избавление от которой может являться нежелательным эффектом при верификации систем, если В РЕАЛИЗАЦИИ все операторы блока `atomic`, выполняются не неделимо и эффект от различного их чередования реально различен.

Выбор по условию

Оператор `if` несколько отличен от обычных условных операторов современных языков программирования. В простом нижеследующем примере можно использовать относительные значения двух переменных `a` и `b` для выбора между двумя опциями:

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

Структура выбора `if` содержит как минимум две последовательности операторов, каждая предваряется двойным двоеточием. Выполняется только одна последовательность из списка возможных (выполнимых). Последовательность выполнима, если ее первый оператор выполним. Первый оператор называется *защитой* (*guard)* или *условием*.

Если выполнимыми оказываются несколько операторов, недетерминировано выбирается какой-либо один. При этом порядок перечисления альтернатив выбора несущественен, он ни на что не влияет. Если все условия невыполнимы, то процесс будет заблокирован, пока хотя бы одно из них не сможет быть выбранным. Ограничения на типы выражений, которые могут быть использованы в качестве защиты, отсутствуют. Следующий пример использует входные операторы.

```
#define a 1
#define b 2

chan ch = [1] of { byte };

proctype A(){
     ch ! a
}

proctype B(){

     ch ! b
}

proctype C(){

     if    :: ch ? a -> option1
           :: ch ? b -> option2
     fi
}

init{
     atomic { run A(); run B(); run C() }
}
```

В примере определены три процесса и один канал `ch`. Первая опция в структуре выбора процесса типа `C` является выполнимой, если канал содержит сообщение-константу `a` (`a` является константой со значением 1). Вторая опция будет выполнимой, если канал содержит сообщение `b` (`b` также является константой). Какой из операторов будет выполнен, зависит от относительных скоростей процессов, которые неизвестны.

Процесс в следующем примере либо увеличивает, либо уменьшает значение переменной `count`:

```
byte count;

proctype counter(){
      if    :: count = count + 1
            :: count = count - 1
      fi
}
```

Поскольку оба выражения в примере всегда выполнимы, то выбор между ними полностью недетерминирован, он не зависит от начального значения переменной `count`.

Цикл

Логическим расширением структуры выбора является *структура повторения* (*цикл*). Модифицируем представленный выше пример таким образом, чтобы получить циклическую программу, которая произвольно меняет значение переменной, увеличивая или уменьшая его:

```
byte count;
active proctype counter(){
      do    :: count = count + 1
            :: count = count - 1
            :: (count == 0) -> break
      od
}
```

Аналогично предыдущему оператору `if`, в операторе `do` для текущего выполнения может быть выбрана только одна опция. После того, как выполнение выбранной опции завершится, управление передается на начало оператора цикла: выполнение структуры повторяется. Обычный способ выхода из цикла – с помощью оператора `break`. Можно считать `break` не оператором, а указанием на то, что цикл завершается, и управление должно быть передано оператору, следующему за ключевым словом `od`. В примере цикл будет прерван, когда `count` будет нулем. В данном примере цикла два других оператора всегда будут выполнимыми, поэтому выбор условия для выхода будет недетерминированным даже при значении `count`, равном 0.

Для того чтобы гарантировать завершение цикла в случае, когда счетчик станет равным нулю, необходимо изменить модель следующим образом.

```
active proctype counter(){
      do    :: (count != 0) ->
              if        :: count = count + 1
                        :: count = count - 1
              fi
            :: (count == 0) -> break
      od
}
```

В структурах выбора и повторений бывает полезно специальное условие `else`. Условие `else` становится выполнимым в операторе выбора, ***только если*** ни одно другое условие в процессе в той же самой точке контроля потоком не выполнимо. Предыдущий пример преобразуем в эквивалентный следующим образом:

```
active proctype counter(){
      do    :: (count != 0) ->
              if        :: count = count + 1
```

```
                        :: count = count - 1
                fi
            :: else -> break
        od
}
```

Условие `else` становится выполнимым именно тогда, когда `!(count != 0)` или, что то же `(count == 0)`, и потому сохраняет возможность выхода из цикла.

Другим способом завершения цикла является использование *безусловного перехода* оператор `goto`. Оператор `goto` всегда является выполнимым, если существует метка, на которую выполняется переход. Это проиллюстрировано в следующем алгоритме Эвклида нахождения наибольшего общего делителя двух положительных чисел:

```
proctype Euclid(int x, y){
      do    :: (x >  y) -> x = x - y
            :: (x <  y) -> y = y - x
            :: (x == y) -> goto done
      od;
done: skip
}
```

где выполнение `goto` вызовет передачу управления на метку `done`. *Метка* может быть поставлена только перед оператором. Здесь метка стоит перед оператором завершения программы. В этом случае пустой оператор `skip` является полезным: он выступает как оператор-заполнитель, который всегда является выполнимым, но не производит никакого эффекта.

1.4 Некоторые примеры

Рассмотрим несколько полезных примеров, демонстрирующих возможности языка Promela. Следующий пример определяет фильтр, который получает сообщения из канала `ch` и делит их на два канала `large` и `small` в зависимости от значения полученных данных. Константа `N` определена как 128 и `size` определена как 16 в двух макро определениях:

```
#define N    128
#define size  16

chan ch    = [size] of { short };
chan large = [size] of { short };
chan small = [size] of { short };

proctype split(){
      short data;
      do    :: ch ? data ->
                  if    :: (data >= N) ->     large ! data
                        :: (data <  N) ->     small ! data
                  fi
            od
      }

init{
      run split()
}
```

Очевидно, что в данной программе процессы заблокируются в начале работы, т.к. `split` будет ждать сообщений из канал `ch`, но этот канал пуст! Ни один из процессов не отправляет

в него сообщения. Введем процесс, который объединяет два потока входных данных в один выходной поток, направляемый в канал ch в произвольном порядке:

```
proctype merge(){
      short data;
      do    ::    if    :: large ? data
                        :: small ? data
                  fi;
                  ch ! data
      od
}
```

Если модифицировать процесс init следующим образом:

```
init{
      ch ! 345; ch ! 12;
      ch ! 6777; ch!32; ch ! 0;
      run split();
      run merge()
}
```

то разделяющий и объединяющий процессы будут выполняться бесконечно. Действительно, изначально init отправляет сообщения в канал ch. Далее процессы merge и split постоянно поставляют сообщения во входные каналы друг друга. Ни один, ни другой процесс не имеет условий, по которым работа может считаться завершенной.

Процессы могут быть смоделированы как *рекурсивные*. Возвращаемое значение передается обратно в вызывающий процесс посредством глобальной переменной или через сообщение. Следующая программа демонстрирует работу рекурсивного процесса.

```
proctype fact(int n; chan p) {
      chan child = [1] of { int };
      int result;

      if    :: (n <= 1) -> p ! 1
            :: (n >= 2) ->  run fact(n-1, child);
                            child ? result;
                            p ! n*result
      fi
}

init{
      chan child = [1] of { int };
      int result;

      run fact(7, child);
      child ? result;
      printf("MSC: result: %d\n", result)
}
```

Процесс fact(n,p) рекурсивно вычисляет факториал n, передавая результат с помощью сообщения своему родительскому процессу p.

2 Описание верифицируемых свойств средствами Promela и SPIN

Подход к верификации на основе метода *model checking* состоит в том, что для модели системы формально проверяется выполнение логической формулы, выражающей некоторое свойство ее “правильного” поведения. Обычно понятие “правильного поведения” включает некоторое множество свойств, которые делают систему полезной для использования, и все эти свойства должны быть проверены последовательно, одно за другим. Свойства распределенных систем традиционно разбиваются на следующие классы:

- свойства достижимости (*reachability*), которые устанавливают, что некоторые специфические состояния системы могут быть достигнуты;
- свойства безопасности (*safety*), устанавливающие, что нечто плохое, нежелательное, никогда не произойдет с системой;
- свойства живости (*liveness*), устанавливающие, что при некоторых условиях нечто “хорошее” в конце концов произойдет при любом развитии процесса;
- свойства справедливости (*fairness*), устанавливающие, что нечто будет выполняться неопределенно часто.

Свойства достижимости являются одними из наиболее часто проверяемых классов свойств параллельных систем: *некоторая конкретная ситуация в процессе функционирования системы случится.* Свойство достижимости естественным образом выражается LTL формулой **EF**φ.

Свойство безопасности гарантирует, что при некоторых условиях некоторая ситуация никогда не может быть достигнута (гарантия того, что *нечто плохое никогда не произойдет*). Свойство безопасности выражается формулой **G**¬φ. Типичные примеры свойства безопасности – взаимное исключение, свобода от дедлоков (блокировок), сохранение инвариантов. Свобода от блокировок является одним из главных требований к параллельным системам: блокировки возникают, когда каждый процесс из группы параллельно работающих процессов ожидает некоторого события (например, каждому процессу для продолжения его работы необходим ресурс, уже захваченный другим процессом, и каждый процесс, захватив ресурс, ждет освобождения другого нужного ему ресурса другим процессом).

Программы, которые ничего не делают, всегда удовлетворяют требованиям безопасности: в них ничего не происходит, поэтому и ничего плохого наступить не может. Очевидно, что требования безопасности в спецификации программ должны сопровождаться требованиями живости, прогресса вычислений системы в нужном направлении. Прогресс в “правильном” направлении обеспечивается свойствами живости. Свойство живости утверждает, что *нечто хорошее в будущем обязательно произойдет*, что выражается формулой CTL **AF**φ, или *нечто хорошее обязательно в будущем будет происходить неопределенно часто*, что выражается формулой LTL **GF**φ.

Во многих случаях выполнение свойств, определяющих корректное функционирование модели, должно проводиться только на тех вычислениях, на которых выполняется некоторое дополнительное условие, которое называется требованием *справедливости*:

$$fairness \Rightarrow свойство$$

Существует две причины возникновения требования справедливости. Первая причина состоит в том, что анализируемая система будет работать в некотором окружении, к функционированию которого мы формулируем эти требования, поскольку только с этими дополнительными требованиями к окружению наша система может выполнять полезную функциональность. Например, справедливый планировщик должен удовлетворять требования предоставления ресурса от каждого процесса. Вторая причина совершенно

другая. Она состоит в том, что в той модели переходов, которая используется при верификации для представления реальных систем, могут существовать нереалистичные, нереализуемые траектории (вычисления), возникшие из-за ограниченных выразительных средств модели. Назовем нереалистичные траектории поведения модели несправедливыми. Справедливые траектории поведения системы в этом случае не нужно специально обеспечивать при реализации: например, реальный канал сам обеспечивает ненулевую вероятность доставки сообщения, но проверку свойств системы нужно выполнять при выполнении условий справедливости – фактически, отбрасывая нереалистичные траектории поведения модели.

При разработке параллельных систем, кроме специфичных требований, которым должно удовлетворять поведение данной конкретной системы, для системы следует проверять еще и общие свойства, гарантирующие отсутствие некорректностей, типичных для параллельных систем. SPIN поддерживает верификацию следующего набора общих свойств, называемых в SPIN базовыми:

- из класса свойств безопасности
  - проверка сохранения локальных инвариантов, описанных при помощи оператора `assert`;
  - проверка на наличие некорректных конечных состояний, т.е. обнаружение взаимных блокировок процессов;
- из класса свойств живости
  - проверка отсутствия бесконечных циклов, не содержащих операторов, помеченных меткой `progress`;
  - проверка отсутствия циклов, с бесконечно частым выполением операторов с меткой `accept`.

2.1 Оператор `assert`

Оператор `assert`, позволяет задавать локальные инварианты, т.е. такие свойства, которые должны выполняться в определенных точках программы

`assert(`любое выражение Promela`)`

Если выражение, стоящее под оператором assert,истинно, то оператор `assert` не производит никакого эффекта. Если выражение будет ложно при симуляции, то SPIN выдаст сообщение «Error: assertion violated». При выполнении базовой верификации свойств безопасности нарушение операторов assert проверяется на всех конечных вычислениях. В SPIN в режиме симуляции могут быть проверены только свойства системы, описанные оператором assert. Все другие механизмы описания свойств, поддерживаемые SPIN, интерпретируются только в режиме верификации.

2.2 Метка заключительного состояния (end)

Когда Promela используется для спецификации модели, которая будет верифицироваться в Spin, пользователь может делать индивидуальные утверждения о поведении, которое моделируется. Например, если код проверяется на наличие взаимных блокировок, верификатор должен уметь отличать нормальные, с точки зрения пользователя, завершающие (заключительные) состояния от не нормальных.

Нормальное заключительное состояние может быть состоянием, когда каждый процесс правильно достиг конца тела описания программы, и все каналы сообщений пусты. Но это не означает, что все процессы кода ДОЛЖНЫ достигнуть конца своего тела программы.

Для того, чтобы сделать понятным для верификатора, что такие альтернативные заключительные состояния допускаются и не являются дедлоком, в модели можно использовать метки заключительного состояния.

В примере, реализующем взаимно исключающий доступ к ресурсу при помощи семафора Дейкстры, добавляя метку, начинающуюся со слова `end` к оператору цикла

```
#define p  0
#define v  1

chan sema = [0] of { bit };

proctype dijkstra(){
     byte count = 1;
endpoint:  do    :: (count == 1) ->
                    sema ! p; count = 0
           :: (count == 0) ->
                    sema ? v; count = 1
     od
}

active[3] proctype user(){
     if    :: sema ? p;
                    /* критическая секция */
                    sema ! v;
                    /* некритическая секция */
     fi
}
```

мы сообщаем системе Spin, что не будет ошибкой (блокировкой процесса), если в конце выполняемой последовательности процесс `dijkstra` не достигнет своей закрывающейся фигурной скобки, а будет ожидать запроса от каких-либо процессов. Процесс `dijkstra()` выполняет здесь роль семафора. Семафор гарантирует, что не более одного пользовательского процесса может войти в критическую секцию. Поскольку процесс dijkstra является обслуживающим, естественно его определить так, чтобы он не знал числа обращающихся к нему клиентских процессов. Поэтому следует рассматривать остановку процесса dijkstra в состоянии, в котором его может запросить клиентский процесс о выполнении последовательности операций p и v над семафором, как корректное состояние завершения процесса.

В модели может быть несколько меток заключительного состояния.

2.3 Метки активного состояния (`progress`)

Так же, как и метки состояния, в процессе могут быть использованы метки активного состояния `progress`. Этими метками помечают операторы, выполнение которых желательно, тем самым усиливая необходимость прогресса в поведении модели. При базовой верификации модели в SPIN проверяется, что любой потенциально бесконечный цикл проходит хотя бы через одну метку `progress`, и таким образом, удовлетворяется свойство *живости* (*liveness*). Если же находится контрпример, нарушающий данное свойство, то SPIN сообщает о существовании непрогрессивного цикла, а следовательно, о возможном голодании процессов. Добавим метку активного состояния к процессу `dijkstra`, моделирующему работу семафора посредством канала `sema`. Операции `P` и `V` над семафором должны выполняться здесь внешними процессами, как `sema ? p` и `sema ! v`:

```
proctype dijkstra()
{     byte count = 1;
endpoint  :     do
      :: (count == 1) ->
progress:  sema ! p; count = 0
      :: (count == 0) ->
            sema ? v; count = 1
      od
}
```

Считая прогрессом постоянную восстребованность семафора внешними процессами, пометим удачную передачу токена семафором меткой `progress`, тогда при базовой верификации SPIN проверит, что в любом бесконечном вычислении семафор будет заблокирован бесконечное число раз.

В процессе допускается использование нескольких меток активного состояния.

2.4 Метка принимающего состояния (accept)

Метки принимающих состояний используются обычно в особом процессе `never` (см. 2.5), хотя PROMELA не ограничивает их использование. Меткой принимающего состояния считается любая метка, начинающаяся префиксом `accept`:

`accept[a-zA-Z0-9_]*`: операторы

Каждая метка принимающего состояния в теле одного процесса должна имеет свое собственное имя, например, `accept`, `acceptance`, `accepting`.

Фактически, эти метки помечают состояния, которые соответствуют принимающим (допускающим) состояниям автомата Бюхи, описывающим все бесконечные “ошибочные”, “нежелательные” траектории проверяемой программы. Ошибочные траектории характеризуются именно тем, что такое принимающее состояние при вычислении проходится бесконечное число раз. Поэтому при верификации проверяется, что в системе не существует вычислений, которые проходят через операторы, помеченные `accept`, бесконечно часто.

2.5 Особый процесс never

Многие из свойств могут быть проверены введением операторов `assert`, меток конечного и активного состояния в тело типа процесса `proctype`. Однако, достаточно трудно определить в модели свойства, подобные данному: *"любое состояние системы, в котором* `p` *истинно, приведет к такому состоянию системы, что истинно* `q`*"*. Проблема состоит в том, что рассматриваемые ранее способы описания свойств плохо предназначены для проверки во всех состояниях системы. Тем более что мы не можем делать никаких предположений об относительных скоростях процессов, т.е. между любыми двумя операторами процесса могут выполниться несколько (неопределенное число) шагов других процессов.

Особый процесс `never` дает возможность задания глобальных инвариантов. Процесс `never` используется для того, чтобы описать такое поведение, которое НЕ ДОЛЖНО произойти в системе. Процесс `never` предназначен лишь для слежения за поведением системы, не оказывая никакого влияния на нее. В этом процессе нельзя объявить переменные или манипулировать каналами сообщений. В нем запрещены всякие действия, способные изменить состояние системы. В модели может быть только один процесс `never`.

Процесс `never` учитывается только при верификации. Он позволяет проверить свойство системы в точности в начальном состоянии и после каждого шага вычисления (т.е. после

выполнения каждого оператора, независимо от того, к какому процессу этот оператор относится).

Простейший вариант использования процесса `never` – проверка выполнения условия `p` на каждом шаге системы:

```
never {
        do    :: !p -> break
              :: else
        od
}
```

Процесс `never` с заданным правилом выполняется НА КАЖДОМ шаге системы. Как только условие p станет ложным, процесс `never` выходит из цикла и прерывается, переходя в завершающее состояние. Завершение процесса `never` интерпретируется как ошибочное поведение анализируемой системы. Если `p` остается истинным, то процесс never остается в цикле, он не завершается, и ошибки в анализируемой системе нет.

Для приведенного свойства можно сформулировать альтернативное описание без процесса `never`. Добавим процесс `monitor`:

```
active proctype monitor(){
        atomic { !p -> assert(false) }
}
```

Здесь процесс с именем `monitor` может инициировать выполнение последовательности операторов, определенных внутри группы `atomic`, в любой точке вычисления системы. Поэтому в любом достижимом состоянии системы, в котором инвариант (в данном случае истинность утверждения p) нарушается, процесс monitor выполняет проверку и сообщает об ошибке с помощью оператора assert, т.е. в этом случае процесс `monitor` решает задачи процесса `never`. Однако, для более сложных темпоральных свойств без процесса `never` не обойтись.

Например, рассмотрим свойство:

Всегда, если `p` *стало истинным, то когда-нибудь в будущем станет истинным q, а p будет оставаться истинным до тех пор, пока* `q` *не станет инстинным.*

Это свойство достаточно просто описывается формулой линейной темпоральной логики (LTL): `G(p → (p U q))` (учтите, что будущее включает настоящее).

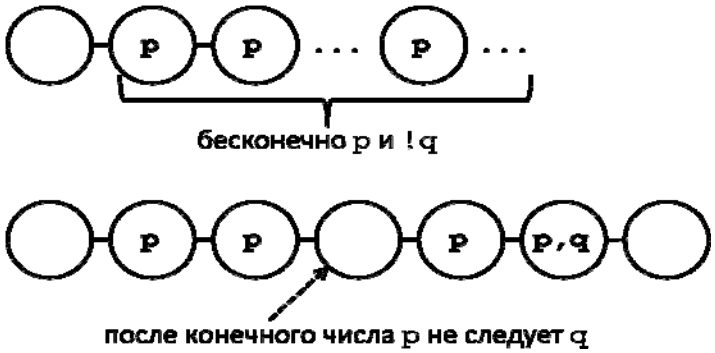


Рис. Примеры ошибочных вычислений для LTL формулы **`G(p → (p U q))`**. Именно такие трассы и будет искать процесс `never`.

При проверке модели нас не интересуют все те вычисления, в которых свойство удовлетворяется, наоборот, для модели проверяется наличие в ней вычислений, на которых свойство нарушается. Все такие “ошибочные” вычисления для нашего случая удовлетворяют

формуле !G(p → (p U q)). При верификации нам нужно выявить среди всех возможных вычислений проверяемой системы хотя бы одно ошибочное вычисление, т.е. такое, в котором p стало истинным, а q осталось ложным на всем вычислении, или p стало ложным до того как, q стало истинным.

Очевидно, что нарушение свойства, где q остается ложным всегда, может случиться только на бесконечных вычислениях. Promela имеет дело с моделями с конечным числом состояний, поэтому бесконечное вычисление появляется только внутри циклов. Мы не можем использовать обычные процессы с assert для обнаружения таких ошибок живости (напомним, assert проверяются только на конечных вычислениях). Воспользуемся процессом never для правильной проверки заданного свойства.

```
never {  /*  !G (p → (p U q))   */
S0:   do      :: p && !q -> break
              :: true
      od;
accept:
      do      :: p && !q
              :: !(p || q) -> break
      od
}
```

В режиме верификации сначала (в начальном состоянии системы) проверяется возможность выполнения первого оператора процесса never. В данном примере выполнение процесса never начинается с метки S0, которая помечает цикл с недетерминированным выбором. Второе условие состоит из одного оператора с выражением true. Очевидно, что эта опция всегда выполнима и не оказывает никакого эффекта на вычисления. Она возвращает процесс never в его начальное состояние. Очевидно, что процесс не может заблокироваться в состоянии S0, т.к. выражение true всегда выполнимо. Наличие true позволяет системе не выполнять проверку свойства, отложив ее на потом.

Первое условие первого цикла do выполнимо, только когда выражение p истинно, а выражение q ложно. Этот случай обнаруживает следующее поведение модели: стало истинно p, но q еще не истинно. Именно с этого состояния может начаться некорректная траектория выполнения анализируемой программы. Она будет некорректной в двух случаях - когда во всех последующих состояниях будет истинно p, а q будет ложно, или если встретится состояние, в котором не будут истинны ни p, ни q.

Если первый оператор цикла выполнится (т.е. найдется состояние, в котором p истинно, а q ложно) то вычисление будет продолжено с метки accept.

В состоянии, помеченном этой меткой, цикл также имеет два условия выбора. Один из путей – вечное пребывание в цикле по условию p истинно, а q – ложно. Если такое вычисление существует, то это соответствует нарушению свойства G (p → (p U q)). Бесконечное повторение этого состояния соответствует ошибочному вычислению. Поэтому это состояние помечено меткой accept. Нарушение будет найдено верификатором как цикл с бесконечным выполнением оператора с меткой accept.

Другое возможное нарушение возникнет, когда p становится ложным до того, как q стало истинным. Этот тип нарушения приведет к завершению процесса never. Завершение процесса never интерпретируется, как найденное ‘ошибочное” поведение, которое совпадает одним из с поведений, которое не должно было бы произойти при выполнении свойства.

Если оба выражения, `p` и `q`, окажутся истинными при выполнении второго оператора, цикла, то ни одно из условий выбора невыполнимо, следовательно, процесс `never` заблокируется. Блокировка процесса `never` не ошибка, а желаемое поведение! Блокировка процесса `never` говорит о том, что он не завершился и не проходил бесконечный цикл с меткой `accept`, т.е. не нашел ошибочных поведений модели.

Важно заметить, что в состоянии `S0` возможна ситуация, когда оба условия (guard) исполнимы. Здесь недетерминированный выбор имеет критическое значение. Если в приведенном выше примере процесса `never` заменить условие с `true` на оператор `else`, то процесс будет реагировать только на первое выполнение выражения `p`, которое сразу должно приводить к выполнению выражения `q`. Наличие недетерминированного условия позволяет сформулировать более сложное условие `G (p → (p U q))`.

Отметим еще раз, что процесс `never` не должен оказывать влияния на поведение системы, поэтому в нем, как видно из примеров, используются лишь операторы условия и `assert`.

Любая метка, начинающаяся словом `accept`, например, `accept1`, `accept_init`, `accepting` и т.п., играет ту же роль в процессе `never`, что и метка `accept`.

Отметим, что построение процесса `never` для обнаружения ошибочных поведений модели не всегда является простым. Spin позволяет описать проверяемые свойства модели формулами линейной темпоральной логики, используя темпоральные операторы F, G и U, определенные пользователем атомарные условия (атомарные предикаты), построенные как булевы ыражения языка Promela, и обычные булевы операторы `&&`, `||` и `!` (и. или и нет). Темпоральный оператор X не может быть использован при построении LTL формул в системе Spin (поскольку понятия “следующее состояние” в системе параллельных процессов нет: в общем случае каждый из параллельных процессов может выполнить независимый шаг). По историческим причинам, в системе Spin темпоральный оператор G представляется парой квадратных скобок [], а темпоральный оператор F – парой угловых скобок <>. Импликация представляется парой символов ‘->’. Указанная выше формула LTL `G (p → (p U q))` будет записана в Spin, как `[](p -> (p U q))`.

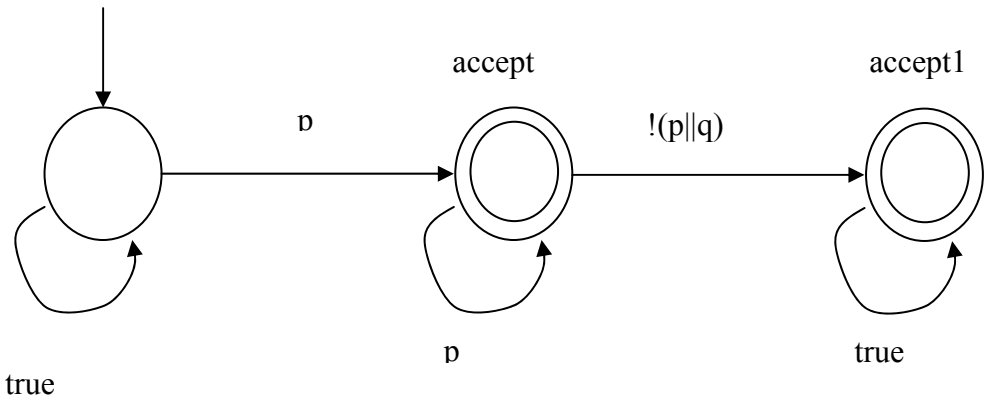


Рис. Недетерминированный автомат Бюхи для формулы `!G(p → (p U q))`, который переходит в заключительное (принимающее) состояние `accept`, если, начиная с любого состояния (переход `true` в состоянии `s0`) встретится бесконечная цепочка состояний, помеченная `p`, или же после конечного числа состояний, помеченных `p`, встретится состояние, в котором не истинны ни `p`, ни `q`.

По введенной темпоральной формуле Ф, выражающей требуемое свойство поведения модели, Spin строит ее отрицание !Ф (формулу, описывающую все “ошибочные” поведения), и по формуле !Ф строит автомат Бюхи $B_{!Ф}$, конечным образом описывающий все такие поведения. Далее, по автомату Бюхи $B_{!Ф}$ Spin строит процесс `never`, который в режиме

верификации будет пытаться обнаружить хотя бы одно “ошибочное” поведение модели аналогично тому, как это описано в приведенном выше примере.

3 Примеры задач для верификации в Spin

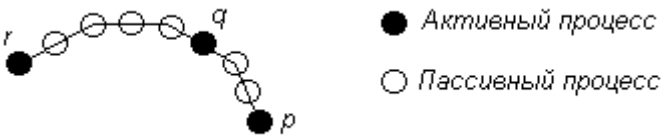
В данном разделе подробно описаны несколько задач, модели которых составлены на языке Promela и верифицированы средствами Spin. Кроме того, здесь изложены основы работы с графической оболочкой XSpin. Ознакомление с этой частью раздела позволит читателям не только самостоятельно моделировать интересующие его задачи, но и проверять корректность разработанных ими моделей при помощи Spin.

3.1 Протокол выбора лидера в однонаправленном кольце

Описание протокола

Рассмотрим протокол выбора лидера в однонаправленном кольце (алгоритм предложен Долевым, Клаве, Роде (Dolev, Klawe, Rodeh) и независимо Петерсоном (Peterson) в 1982 году) [2, 3]. До создания этого алгоритма считалось, что при выборе лидера в однонаправленном кольце количество сообщений не может быть меньше, чем $O(N^2)$, где N – число узлов в кольце. В данном алгоритме достигается количество сообщений 2Nlog2N+O(N). Рассмотрим этот алгоритм.

Дано N распределенных узлов с некоторыми весами. Узлы асинхронно взаимодействуют только с соседями, образуя однонаправленное кольцо. Количество узлов фиксировано и не меняется. Порядок сообщений в каналах не меняется. Требуется построить протокол – набор локальных правил для каждого узла, которые позволят получить глобальный результат – каждому узлу определить единственного лидера (например, узел с наибольшим весом).



Изначально алгоритм был разработан для двунаправленных колец, и при обходе круга активный процесс сравнивал себя с двумя соседними активными процессами по часовой стрелке и против нее. Таким образом, если его номер являлся локальным максимумом, он оставался в круге, иначе он становился пассивным. Активным узлом считается узел, который создает сообщения с новой информацией. Пассивный узел лишь передает полученные сообщения соседу, никак их не изменяя.

В ориентированных кольцах сообщения можно посылать только по часовой стрелке, что затрудняет получение номера соседнего активного процесса, находящегося в этом направлении. Будем считать, что направление обхода в кольце по часовой стрелке, и только по этому направлению посылаются все сообщения.

В алгоритме Долева, Клаве, Роде на каждом активном узле хранятся два параметра: локальный максимальный вес `maximum` и параметр предыдущего активного соседа, находящегося против часовой стрелки `neighbourR`. Локальный максимальный вес вычисляется по данным от двух предшествующих против часовой стрелки активных соседей.

Алгоритм состоит из повторяющихся раундов, на которых обновляются значения параметров на активных узлах. За один раунд не менее половины активных узлов переходит в пассивное состояние. Именно за счет этого правила достигнуто сокращение общего количества сообщений, необходимых для выбора лидера. Каждый из раундов состоит из двух шагов: на первом шаге A1 активные узлы получают сообщения типа `one`, а на втором шаге A2 – типа `two`.

Пассивные узлы только передают по кольцу полученные сообщения. Опишем последовательность действий активного узла. В начале работы алгоритма все узлы являются активными:

A0. `maximum :=` собственный вес. Послать сообщение `one(maximum)` ближайшему соседу.

A1. Если пришло сообщение `one(q)`, то:

1. Если `q != maximum`, то присвоить `neighbourR значение q` и послать сообщение `two(neighbourR)`.

2. Иначе, `maximum` и есть глобальный максимум. Лидер найден.

A2. Если пришло сообщение `two(q)`, то:

1. Если `neighbourR` больше и `q`, и `maximum`, то присвоить `maximum значение neighbourR` и послать сообщение `one(maximum)`.

2. Иначе узел становится пассивным.

На каждом узле чередуются раунды A1, A2, A1, A2 и т.д., пока не будет найден лидер.

Этот протокол неочевиден, его непросто понять и, тем более, невозможно гарантировать без глубокого анализа, что он будет выполнять задачу нахождения единственного лидера множества процессов при любом числе процессов и любой расстановке их весов. Такая ситуация характерна для параллельных взаимодействующих процессов: мы имеем точное описание того, что делает каждый процесс, но из этого описания невозможно заключить, выполняется ли некоторое глобальное свойство, которое должно быть обеспечено при работе всех процессов. Именно поэтому этот пример используется здесь для демонстрации анализа (симуляции и верификации), который может быть выполнен с помощью пакета Spin.

Описание модели протокола на языке Promela

Каждому узлу ставится в соответствие процесс типа `p`. Запускается N таких процессов. У каждого процесса есть один входящий канал `in`, через него приходят сообщения от соседа против часовой стрелки, и один исходящий канал `out`, в который процесс посылает сообщения ближайшему соседу по часовой стрелке.

Процесс `p` является активным, если у него флаг `Active` установлен в `true`. Иначе `p` является пассивным и просто пропускает через себя все получаемые сообщения. Активный процесс посылает свой текущий локальный максимум следующему активному процессу, и получает текущий локальный максимум предыдущего активного процесса, используя сообщения типа `one`. Полученный номер сохраняется в переменной `neighbourR`, и если процесс не выбывает из числа активных, он будет текущим локальным максимумом `p` в следующем круге. Чтобы определить, остается ли номер `neighbourR` в кольце, его сравнивают с максимальным номером `maximum`, пришедшим в этот процесс, и активным номером, полученным в сообщении типа `two`. Процесс `p` посылает сообщение `two(neighbourR)`, чтобы следующий активный процесс мог провести такое же сравнение. Исключение возникает, когда `neighbourR = maximum`: в этом случае остался один активный процесс, и об этом сообщается всем процессам в сообщении `winer(nr)`.

Для перевода модели на язык описания Promela сделаны следующие допущения:

- пусть каналы между процессами будут не бесконечными, а с глубиной, например, 10, т.е. как только в канале накопится 10 сообщений, произойдет переполнение канала, и следующие сообщения не смогут в него поступать до того, как освободится место;
- ограничимся количеством процессов равным 5, т.к. в языке Promela необходимо задать какое-либо произвольное, но конкретное число процессов;
- допустим, что все процессы подключаются одновременно, т.е. никакой процесс не может включиться в выбор лидера на более поздней стадии.

Модель алгоритма выбора лидера на языке Promela

```
1 #define N 5                                   /* количество процессов */
2 #define I 3                                   /* порядковый номер процесса,
                                                  которому будет присвоено наименьшее значение */
3 #define L 10                                  /* размер буфера (>= 2*N) */
4
5 mtype = {one, two, winner};                   /* 3 возможных типа сообщений */
6 chan q[N] = [L] of {mtype, byte};              /* массив асинхронных каналов с буфером
                                                  размером L */
8 byte nr_leaders = 0;                  /* переменная, равная количеству процессов,
                                                  считающих себя лидерами кольца */
9
10 proctype node (chan in, out; byte mynumber)  /* определение процесса */
11 { bit Active = 1,                            /* значение 1 переменной Active показывает,
                                                  что данный процесс активен */
   know_winner = 0;                             /* переменная know_winner показывает,
                                                  что процесс еще не знает, кто лидер */
12    byte nr, neighbourR, maximum = mynumber;
                                                  /* объявляется три переменных типа
                                                  byte, одной присваивается значение
                                                  mynumber */
13
14    xr in;                                    /* запрашивается эксклюзивный доступ
                                                  к получению сообщений из канала in */
15    xs out;                                   /* запрашивается эксклюзивный доступ к посылке
                                                  сообщений по каналу out */
```

/* В Promela, в случае если по алгоритму процесс будет единственным осуществляющим запись в канал (или чтение из канала), для уменьшения числа состояний рекомендуется запросить эксклюзивный доступ на запись (или на чтение), что здесь и сделано. */

```
16
17    printf("MSC: %d\n," mynumber);            /*  отладочная печать */
18    out ! one(mynumber);                      /* посылаем сообщение типа one с параметром
                                                  mynumber – номером данного процесса  */
19    end: do                                   /*  метка end показывает, что не будет ошибкой,
                                                если процесс в итоге остается в этом цикле вечно */
20    :: in?one(nr)->                           /* если получаем сообщение типа one с номером,
                                               то записываем номер в переменную nr и смотрим: */
21          if
22          :: Active ->                        /*  если процесс активен, то: */
23                if
24                :: nr != maximum ->           /*  если максимум еще не найден*/
25                      out ! two(nr);          /*  то передаем сообщение типа two
                                                  с полученным номером соседа  */
26                      neighbourR = nr
27                :: else ->
28                                              /* если же максимум найден (т.е. возникла
                                                  ситуация, когда остался 1 активный процесс  )*/
29                      assert(nr == N);/*  то убедимся, что этот максимум равен
                                                  количеству процессов  */
30                      know_winner = 1;  /*  и отметим, что теперь известен лидер  */
31                      out ! winner,nr;  /*  передадим сообщение типа winner
                                                  с номером узла, который мы считаем лидером  */
32                fi
33          :: else ->                          /*   если же процесс не активен*/
34                out ! one(nr)                 /*  то просто пропускаем сообщения дальше*/
35          fi
36
37    :: in?two(nr) ->                          /* если получаем сообщение типа two,
                                                  то записываем полученный номер
```

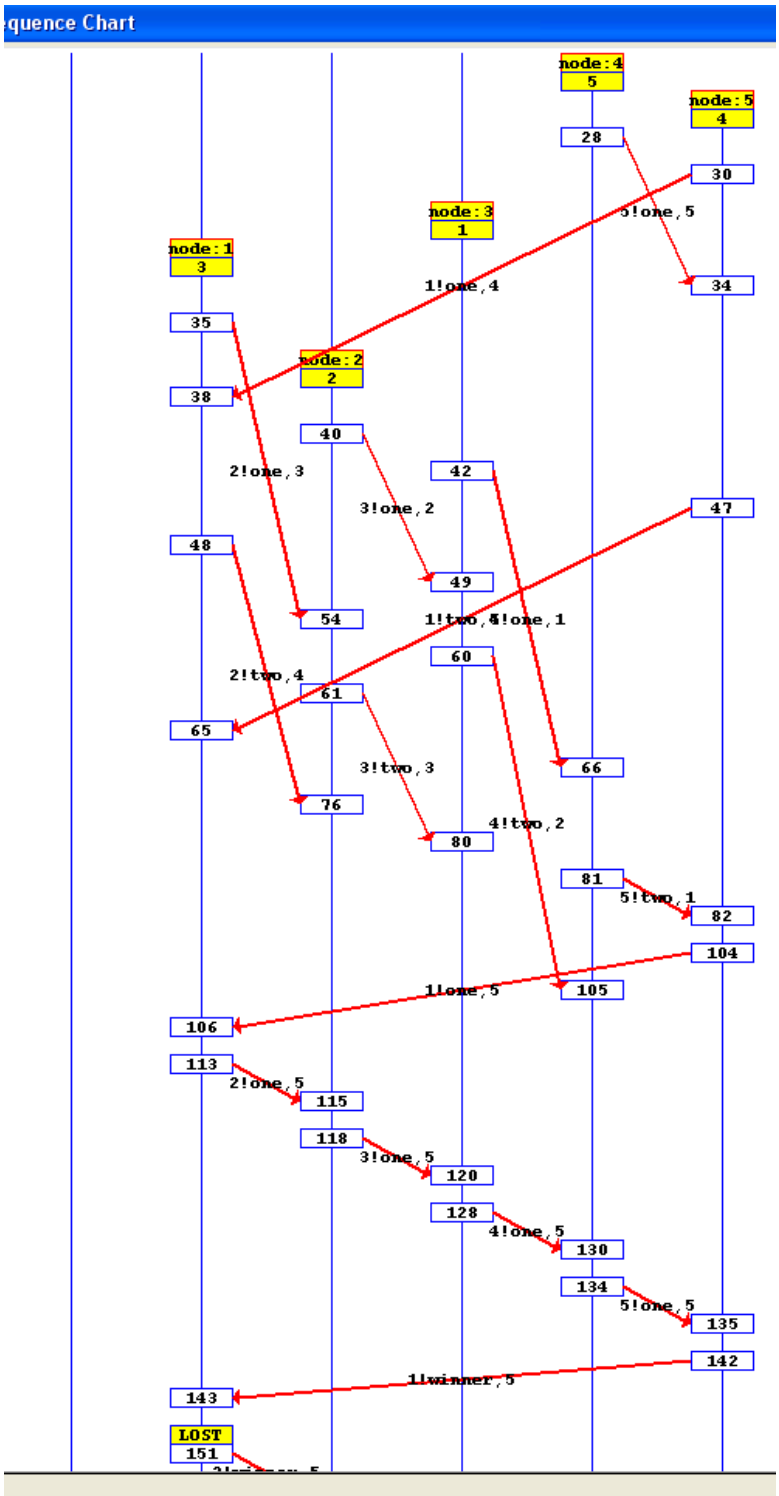
```
                                        в переменную nr */
38          if
39          :: Active ->                /* если процесс активен, то сравниваем его номер
                                        с соседним активным слева и
                                        локальным максимумом   */
40                if
41                :: neighbourR > nr && neighbourR > maximum ->  /*  если номер
                                        оказывается новым локальным максимумом   */
42                      maximum = neighbourR;   /* сохраняем значение нового
                                        максимума   */
43                      out ! one(neighbourR)   /*   передаем номер дальше  */
44                :: else ->
45                      Active = 0        /*   иначе процесс становится пассивным   */
46                fi
47          :: else ->                  /*   если же процесс не активен*/
48                out ! two(nr)         /*  то просто пропускаем сообщения дальше*/
49          fi
50    :: in ? winner,nr ->          /* если получили сообщение о том,
                                        что лидер выбран, то:  */
51          if
52          :: nr != mynumber ->    /*   если текущий процесс не лидер*/
53                printf("MSC: LOST\n");
54          :: else ->                  /*   если же, наоборот, текущий процесс
                                        является лидером*/
55                printf("MSC: LEADER\n");
56                nr_leaders++;         /*   количество лидеров увеличилось*/
57                assert(nrleaders == 1)   /*   проверим, что лидеров в системе только 1*/
58          fi;
59          if
60          :: know_winner              /*  если процесс уже знал о том, что лидер выбран,
                                        то этот процесс его и назначил. Значит, мы уже
                                        обошли все кольцо и нужно заканчивать работу*/
61          :: else -> out!winner,nr    /*  иначе передадим сообщение о лидере дальше*/
62          fi;
63          break
64    od
65 }
66
67 init {                               /* инициализирующий процесс */
68    byte proc;
69    atomic {                          /* одной атомарной  операцией запускаем N копий
                                        процесса node*/
70          proc = 1;
71          do
72          :: proc <= N ->
73                run node (q[proc-1], q[proc%N], (N+I-proc)%N+1);
                                        /*  каждые 2 соседних процесса связывают
                                        2 уникальных канала in и out */
74                proc++
75          :: proc > N ->
76                break
77          od
78    }
79 }
```

Пример симуляции модели выбора лидера

Процесс с номером 5 первым посылает свой номер по каналу соседнему процессу [28 -> 34].

Все остальные процессы делают то же самое, прежде чем начать принимать соседние значения [30 -> 38; 35 -> 54; 40 -> 49; 42 -> 66].

После того, как 4-ый процесс отправил свой номер следующему (3-ему) [30 -> 38], он получает номер от 5-го процесса (для него предыдущего) [28 -> 34]. Т.к. полученное сообщение типа `one` и этот номер не равен сохраненному максимуму (4-ем), то предаем этот номер в сообщении типа `two` следующему процессу [47 -> 65].



То же самое делают и все остальные процессы [48 -> 76; 61 -> 80; 60 -> 105; 81 -> 82], т.к. максимум пока не может быть достигнут.

Когда процесс 4 получает сообщение типа `two` с номером 1 [81 -> 82], он сравнивает сохраненный номер активного соседа (5, полученный из 28 -> 34) и свой максимум (4, т.к. он еще не менялся) с полученным значением. Получается, что номер активного соседа больше в обоих случаях => это локальный максимум => сохраняем максимум и следующим сообщением посылаем его [104 -> 106].

Процесс 3, получив сообщение типа `two` со значением 5 [47 -> 65], сравнивает сохраненный номер активного соседа (4) и свой максимум (3) с полученным значением. Получается, что номер активного соседа меньше => процесс 3 становится пассивным и дальше просто пропускает через себя все сообщения [104 -> 106] -> [113 -> 115].

То же самое происходит и с процессом 2 после [48 -> 76], и с процессом 1 после [61 -> 80], и, наконец, с процессом 5 после [60 -> 105].

Таким образом, сообщение типа `one` со значением 5 возвращается обратно [134 -> 135] к процессу №4, пославшему это сообщение. Процесс проверяет, что значение в сообщении равно максимуму и сообщает остальным, что лидер выбран [142 -> 143].

После окончания симуляции в окне *Simulation Output* выведены все сообщения от *Spin*. В данных сообщениях для каждого шага симуляции показано, какой процесс выполнялся и какая строчка описания протокола на Promela была выполнена.

В окне *log* (внизу основного окна программы) выводятся строчки:

```
.starting simulation.
spin -X -p -v -g -l -s -r -n1 -j0 pan_in
.at end of run.
```

Они показывают команды, которые были синтезированы Xspin, для выполнения симуляции с указанными параметрами.

Для закрытия окон симуляции нажмите кнопку *Cancel* на панели *Simulation Output*.

Верификация базовых свойств

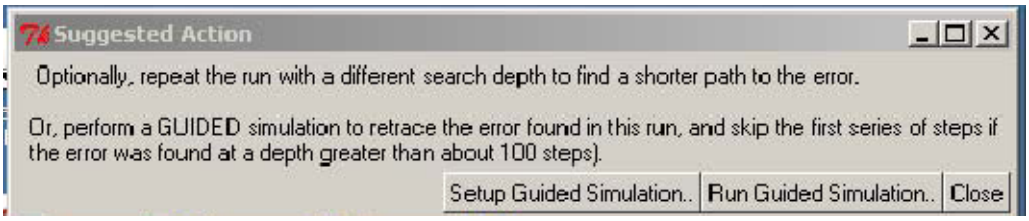
Запустите верификацию базовых свойств, нажав кнопку *Run* на панели *Set Verification Parameters*. Верификация модели `leader` должна пройти без ошибок, и в открывшемся окне *Verification Output* выводится сообщение "`errors: 0`". Закройте окно верификации.

Введем искусственную ошибку в программу, добавив в строку 22 оператор `assert(false)`.

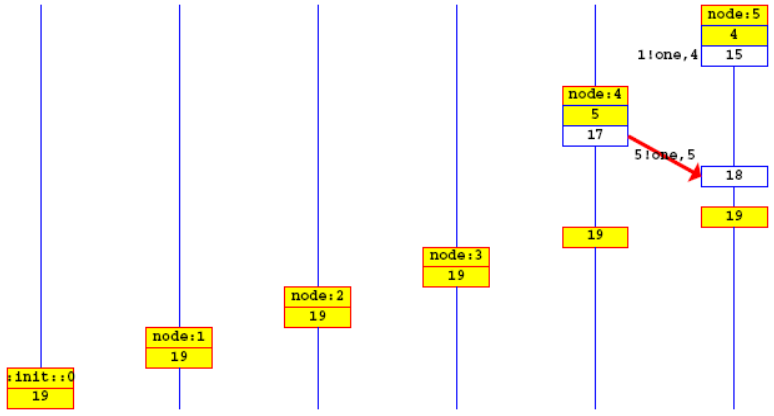
В результате строка будет выглядеть:

```
:: Active -> assert(false);
```

Снова запустим верификацию. Теперь SPIN обнаружит ошибку, и предложит посмотреть контрпример, ее демонстрирующий *Run Guided Simulation (Запустить управляемую симуляцию)*.



Если выбрать изменение установок управляемой симуляции *Setup Guided Simulation*, то XSpin снова панель задания параметров симуляции, все опции в которой, кроме одной остались без изменений. XSpin установил в *Simulation style Guided Simulation* вместо *Random Simulation*.



Если выбрать *Run Guided Simulation*, и затем *Run* в окне *Simulation Output*, то XSpin покажет путь, который привел к ошибке.

В этот раз симуляция завершается в момент, когда было достигнуто нарушение в `assert`, а не когда все процессы дошли до своего конечного состояния.

После завершения симуляции закройте все окна симуляции, нажав кнопку *Cancel* в *Simulation Output Panel*. Удалите добавленный оператор `assert` (т.е. строка 22 будет выглядеть следующим образом `:: Active ->`).

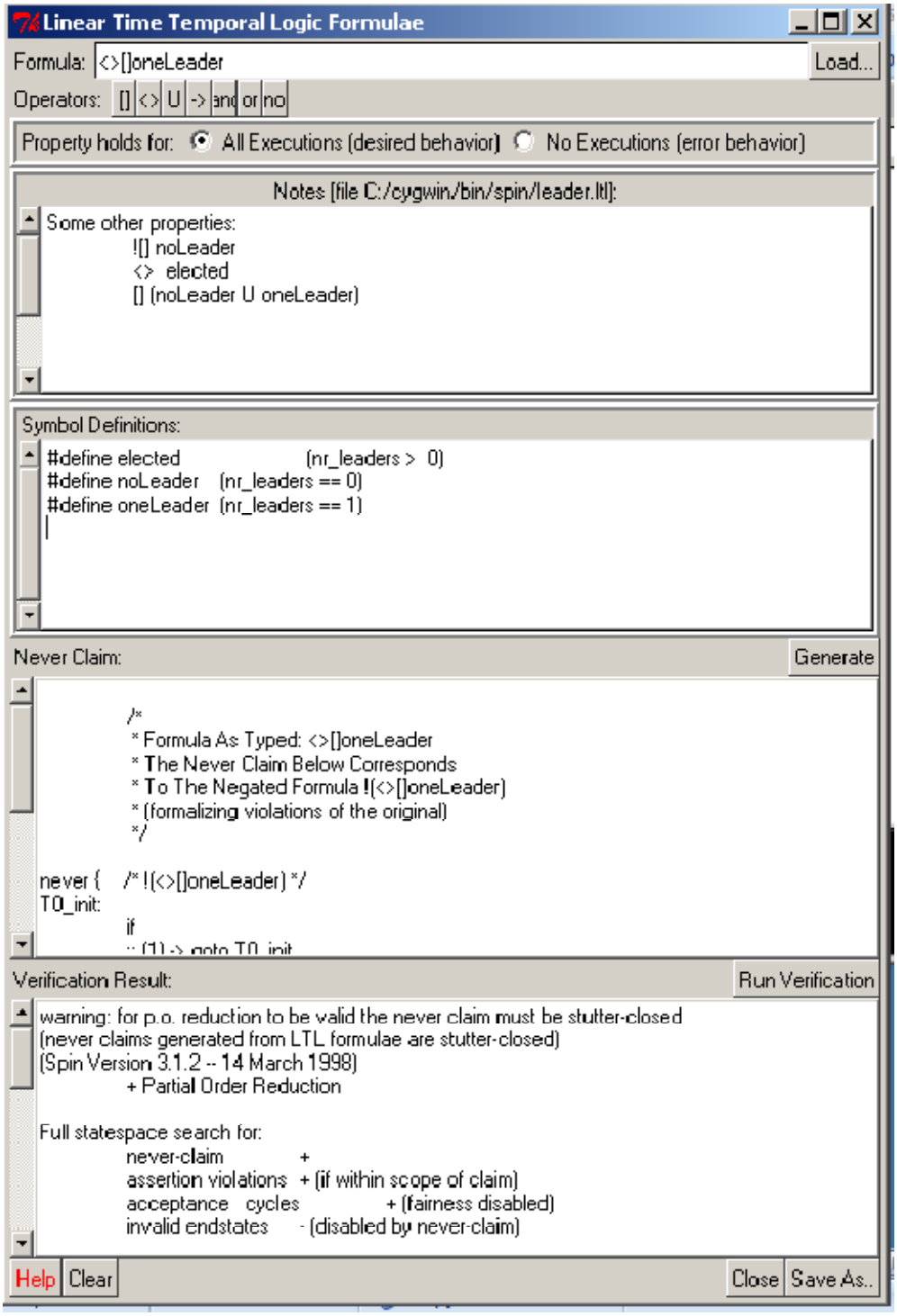
Проверка LTL формул

Пользователь может определить свои специфичные требования к модели в SPIN, выразив их в виде LTL формул. Обозначение темпоральных операторов в «Редакторе LTL формул» (*LTL Property Manager* в меню *Run*):

<>q	Fq	*хотя бы раз в будущем* q
[]q	Gq	*всегда в будущем* q
w U q	w U q	w *до тех пор пока* q

где q – атомарный предикат.

По умолчанию XSpin пытается считать файл с расширением .ltl, и тем же именем, что и файл на Promela, загруженный в основном окне. В нашем случае будут загружены данные из leader.ltl.



1. Проверим требование, что ***лидер должен быть только 1***, т.е. что в нашей модели не может быть выбрано два и более лидеров.

Введем новую формулу в окно *Formula*:

```
[] noMore
```

Данной свойство означает, что выражение, заданное с помощью `noMore`, должно быть всегда истинно. Можно задать, является ли выполнение данной формулы желаемым поведением системы или ошибочным: в первом случае выбирается в *Property holds for* пункт *All Executions*, иначе - *No Executions*. Так как выполнение свойства является желаемым, выберите в *Property holds for* пункт *All Executions.*

Атомные предикаты, подобные `noMore`, задаются в окне *Symbol definition box*.

```
#define noMore (nr_leaders <= 1)
```

Нажмите кнопку *Generate*. В окне *Never Claim* для заданной LTL формулы генерируется особый процесс never, с включенным в него ОТРИЦАНИЕМ формулы.

```
      /*
       * Formula As Typed: []  notMore
       * The Never Claim Below Corresponds
       * To The Negated Formula !([]  notMore)
       * (formalizing violations of the original)
       */

never {    /* !([]  notMore) */
T0_init:
      if
      :: (! ((notMore))) -> goto accept_all
      :: (1) -> goto T0_init
      fi;
accept_all:
      skip
}
```

Результаты верификации будут положительные, в том смысле, что данная формула выполняется на данной модели:

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

State-vector 200 byte, depth reached 205, errors: 0
       97 states, stored
        1 states, matched
       98 transitions (= stored+matched)
       12 atomic steps
hash conflicts: 0 (resolved)

unreached in proctype node
      line 53, "pan.___", state 28, "out!two,nr"
      (1 of 49 states)
unreached in proctype :init:
      (0 of 11 states)
unreached in proctype :never:
      line 104, "pan.___", state 8, "-end-"
      (1 of 8 states)
```

SPIN выводит достаточно много дополнительной информации (число состояний, вектор глобального состояния, недостижимые участки кода и т.п.), которая необходима для оптимизации модели.

Таким образом, мы убедились, что лидер выбирается всегда только 1. Ту же самую проверку можно сделать и с помощью оператора `assert` и базовой верификации (в приведенном выше тексте программы это уже сделано):

```
57 assert(nrleaders == 1)
```

Строчка идет в тексте программы сразу после добавления нового лидера, для того, чтобы Spin проверил, что добавленный лидер оказывается единственным в модели.

2. Проверим, что ***лидер, в конце концов, будет выбран*** при помощи LTL формулы

```
 <>  []  oneLeader,
```

где в качестве атомного предиката задано:

```
#define oneLeader     (nr_leaders == 1).
```

Эта формула означает, что когда-нибудь наступит момент, после которого количество лидеров всегда будет 1, т.е. лидер будет выбран. Свойство на этой модели выполняется.

3. Проверим, что ***номер выбранного лидера всегда будет максимальным*** (т.е. в данной программе равным количеству процессов).

Это свойство легче всего проверить с помощью добавления оператора `assert(nr == N)` в том месте кода, где считается, что лидер найден (см. строчку 29 в коде), `nr` – номер процесса, которого мы начинаем считать лидером, а `N` – число процессов в модели. В рассматриваемой модели свойство выполняется.

Немного модифицируем модель так, чтобы не все процессы участвовали в выборе. Введем глобальные переменные: `election_started = 0` и `first_message = 1`. Если раньше у нас каждый процесс в начале работы был активен и отправлял свой идентификатор в канал, то теперь все процессы в начале неактивны `Active` = 0 и ничего не отправляют в канал

(уберем 18-ую строчку). Лишь один, недетерминировано выбранный процесс, инициирует протокол, установив `election_started = 1`(между 19 и 20-ой строками добавим):

```
:: atomic {
(!election_started && !Active) ->
/* Один процесс может начать выборы  послав первое сообщение msg */
/* и став активным */
election_started = 1;
out ! one(mynumber);
first_message = 0;
Active = 1}
}
```

Далее один из неактивных процессов, первым обнаруживший, что `first_message=1`, может вступить в выборы, а потом установить `first_message=0` (добавим следующий код между 20 и 21 строчками и 37 и 38 строчками):

```
if
:: (first_message && !Active) ->
     if
     :: Active = 1;   /* присоединиться к выборам */
     out ! one(mynumber)
     :: skip          /* не присоединяться к выборам */
     fi
:: else
fi;
first_message = 0;
```

Проверим корректность измененной модели относительно свойства, что номер лидера всегда будет максимальным:

```
pan: assertion violated (nr==5) (at depth 45)
pan: wrote pan_in.trail
(Spin Version 4.2.7 -- 23 June 2006)
Warning: Search not completed
     + Partial Order Reduction

Full statespace search for:
     never claim          - (not selected)
     assertion violations +
     cycle checks         - (disabled by -DSAFETY)
     invalid end states   - (disabled by -E flag)

State-vector 196 byte, depth reached 45, errors: 1
      33 states, stored
       0 states, matched
      33 transitions (= stored+matched)
      13 atomic steps
hash conflicts: 0 (resolved)
```

Из результатов видно, что в поведении модели существуют пути, такие чтооператор `assert(nr == N)` не выполнился (`assertion violated`). Почему это произошло? Потому что модель изменена таким образом, что теперь не все процессы участвуют в голосовании, т.е. будет выбираться не максимальный номер из существующих, а максимальный из участвующих.

Остальные же свойства, проверявшиеся ранее, и на измененной модели останутся истинными.

3.2 Задача о фермере, волке, козе и капусте

Описание задачи

В качестве второго примера рассмотрим задачу, хорошо известную всем со школьных лет: фермер, волк, коза и капуста находятся на левом берегу реки. Фермер хочет переправить волка, козу и капусту на правый берег. В лодку вместе с собой фермер может взять лишь одного из них (либо волка, либо козу, либо капусту). Он может плыть в лодке и один. Однако если волк останется наедине с козой на одном берегу, то он ее может съесть, а коза, оставшись вместе с капустой, поедает капусту. Существует ли такая последовательность переправ фермера, чтобы доставить всех на правый берег целыми и невредимыми?

В данном разделе мы покажем, что с помощью системы Spin мы можем получить решение этой задачи: сформулировав в качестве проверяемого свойства модели то, что ***не существует решения*** с желаемыми свойствами. Как результат верификации Spin выдаст котрпример – именно ту траекторию поведения системы <фермер-волк-коза-капуста>, которая нарушает проверяемое свойство. Этот контрпример и будет являться решением задачи.

Модель задачи о волке, козе и капусте на языке Promela

В данной модели достаточно одного единственного процесса `river`. За местоположение каждого из участников задачи пусть отвечает своя битовая переменная, таким образом, всего будет четыре таких переменных (`f`, `w`, `g`, `c`). Будем считать, что значение переменной равно 0, если соответствующий участник находится на левом берегу, и равно 1 в случае, если участник находится на правом берегу. Применим недетерминированный подход к моделированию задачи о волке, козе и капусте. Недетерминировано выбираются партнеры фермера по переправе из числа тех участников задачи, которые находятся с ним на одном берегу. Например, изначально, партнерами фермера могут быть или волк, или коза, или капуста, или же он может поехать в одиночестве. Повторяя такой выбор в цикле, фермер, в принципе, бесконечно будет перевозить своих подопечных с левого берега на правый и с правого берега на левый. Поскольку по условию задачи все должны быть переправлены на правый берег, то следует добавить условие выхода из цикла:

```
(f==1) && (f == w) && (f ==g) && (f == c)
```

Это условие будет истинным, когда все битовые переменные равны 1 (т.е. волк, коза, капуста, фермер оказались на правом берегу). С его помощью мы можем остановить выполнение цикла всех возможных переправ.

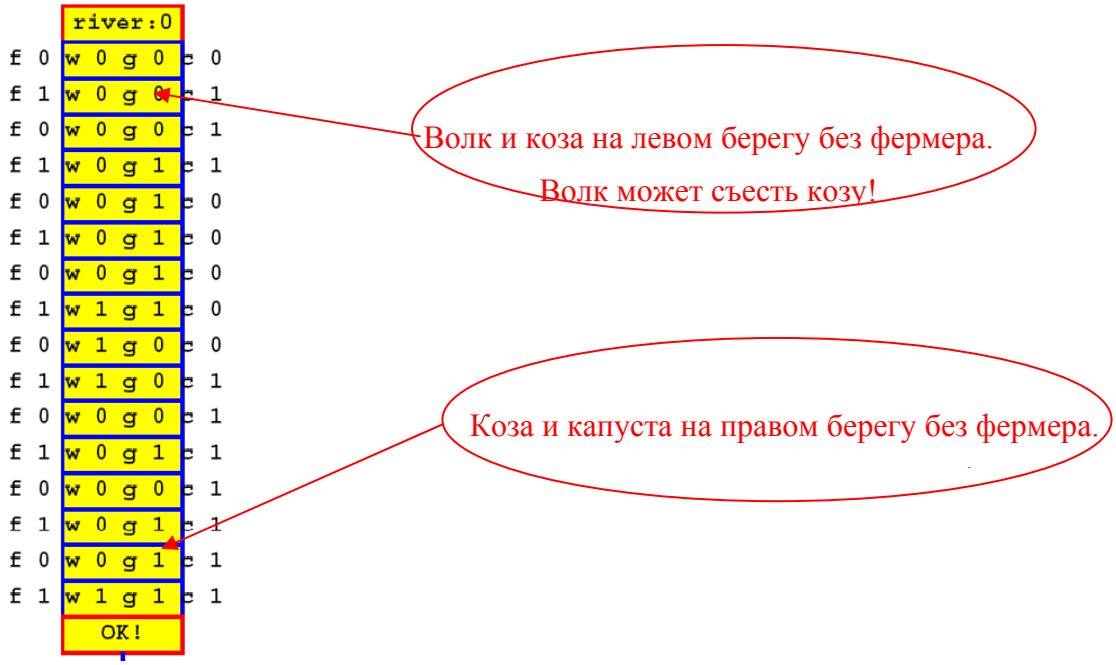
В результате получится следующая модель задачи на языке Promela:

```
/* Задача о фермере (f), волке (w), козе (g) и капусте (c) */

/* Булевы переменные, используемые в LTL формулах */
bool all_right_side, g_and_w, g_and_c;

active proctype river()
{
        /* Вначале волк, коза, капуста и фермер находятся на левом берегу реки */
        bit   f = 0,        /* положение фермера*/
              w = 0,        /* положение волка*/
              g = 0,        /* положение козы*/
              c = 0;        /* положение капусты*/
```

```
        all_right_side    = false;
        g_and_w            = false;
        g_and_c            = false;

        printf("MSC: f %c w %c g %c c %c \n",  f, w, g, c);

        do
        :: (f==1) && (f == w) && (f ==g) && (f == c) ->
              all_right_side = true;  /* все на правом берегу */
              break;                          /* Завершаем цикл*/
        :: else ->
              if                          /* недетерминированный выбор */
                  :: (f == w) ->          /* фермер и волк на одном берегу реки */
                          f = 1 - f; /* фермер переправляет волка */
                          w = 1 - w;
                  :: (f == c) ->          /* фермер и капуста на одном берегу реки */
                          f  = 1- f; /* фермер переправляет капусту */
                          c = 1- c;
                  :: (f == g) ->           /* фермер и коза на одном берегу реки */
                          f  = 1-f;   /* фермер переправляет козу */
                          g = 1-g;
                  :: (true) ->            /*  в любой ситуации */
                          f = 1 - f;   /*  фермер может пересечь реку один */
              fi;

              printf("MSC: f %c w %c g %c c %c \n",  f, w, g, c);

              if
              /* проверяем выполнение ограничений */
              /* съела ли коза капусту? */
                  ::(f != g && g == c ) ->
                          g_and_c = true;
              /* съел ли волк козу? */
                  ::(f != g && g == w ) ->
                          g_and_w = true;
                  ::else ->
                          skip
             fi
        od;

        printf("MSC: OK!\n")
}
```

Поиск решения задачи средствами Spin

При запуске симуляции обычно получаются реализации, где волк может съесть козу, а коза – капусту, и более того, такие возможности "поедания" могут встретиться несколько раз. Эти результаты связаны с тем, что на поведение участников не накладывались никакие ограничения. Приведем пример одного из прогонов симуляции. При выводе используются соглашения, принятые ранее: значение переменной, отражающей местоположение участника равно 0, если участник находится на левом берегу и 1, если на правом.



Обычно Spin используется как средство верификации модели. Однако в данном случае представляет интерес задача: обнаружить во всем разнообразии переправ такую, что фермер переправит все свое имущество (волка, козу и капусту) на правый берег реки в целости. Зададим следующую LTL формулу:

```
<>  fin && [] ( wg  && gc )
```

где значения предикатов таковы (`all_right_side`, `g_and_w`, `g_and_c` булевские переменные, определенные в тексте программы):

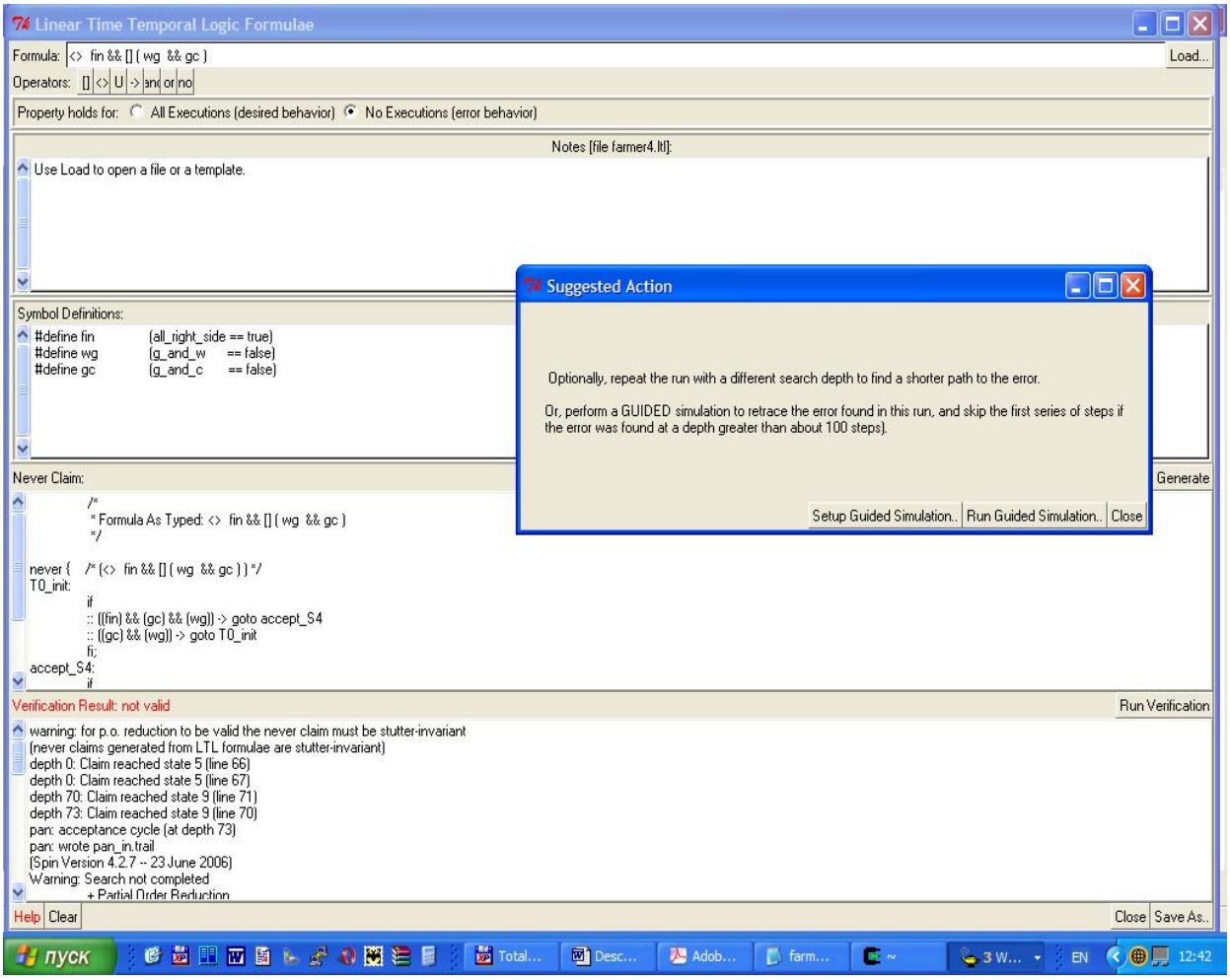
```
#define fin     (all_right_side == true)
#define wg      (g_and_w        == false)
#define gc      (g_and_c        == false)
```

Предположим, что не существует ни одной реализации (устанавливается флаг *No Executions* в пункте *Property holds for* окна проверки LTL формул), в которой выполняется заданное правило: "Когда-нибудь в будущем все участники задачи окажутся на правом берегу, и при этом всегда будет выполняться условие, что волк не съест козу, и коза не съест капусту".

При верификации Spin выдаст ошибку, сообщая тем самым, что он обнаружил ветку поведения процесса `river`, в которой заданное условие выполняется.

```
State-vector 20 byte, depth reached 73, errors: 1
      56 states, stored (57 visited)
       8 states, matched
      65 transitions (= visited+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
```

Spin предложит перейти к управляемой симуляции (*Run guided simulation*):



	river:0	
f 0	w 0 g 0	c 0
f 1	w 0 g 1	c 0
f 0	w 0 g 1	c 0
f 1	w 1 g 1	c 0
f 0	w 1 g 0	c 0
f 1	w 1 g 0	c 1
f 0	w 1 g 0	c 1
f 1	w 1 g 1	c 1
	OK!	
	72	

При прогоне управляемой симуляции будет отображено одно из возможных решений задачи:

1. сначала все вчетвером находятся на левом берегу реки `f = 0 w = 0 g = 0 c = 0`
2. фермер отвозит козу на правый берег, волк и капуста остались на левом берегу, `f = 1 w = 0 g = 1 c = 0`
3. фермер возвращается в одиночестве на левый берег, `f = 0 w = 0 g = 1 c = 0`
4. фермер забирает волка и переплывает с ним на правый берег, `f = 1 w = 1 g = 1 c = 0`
5. фермер с козой переправляется на левый берег, `f = 0 w = 1 g = 0 c = 0`
6. взяв капусту, фермер плывет на правый берег, `f = 1 w = 1 g = 0 c = 1`
7. оставшиеся шаги посвящены переправе козы на правый берег.

Таким образом, система верификации Spin СГЕНЕРИРОВАЛА решение задачи о волке, козе и капусте, как контрпример свойства «интересующего нас решения задачи НЕ СУЩЕСТВУЕТ”.

3.3 Криптографический алгоритм Нидхама-Шредера

Описание алгоритма

В данном разделе рассматривается алгоритм аутентификации – построения доверительных отношений между двумя партнерами A и B, широко применяемом в разных пакетах, например, в Kerberos. Алгоритм, предложенный в 1978 Нидхамом (Needham) и Шредером (Schroeder), использует криптографию публичных ключей. Каждый участник имеет пару ключей. Один ключ – открытый или публичный, известный всем другим участникам, другой ключ – закрытый (хранящийся скрытно), известный только самому владельцу. Связь между двумя ключами такова, что информация, зашифрованная одним ключом, может быть расшифрована только вторым ключом в паре. Таким образом, отправитель может быть уверен в том, что сообщение, зашифрованное им публичным ключом A, сможет прочесть лишь сам A.

В алгоритме аутентификации Нидхама-Шредера стороны обмениваются сообщениями, состоящими из открытой части и зашифрованной части. В открытой части сообщения содержатся сведения об отправителе, получателе сообщения, и номер сообщения согласно алгоритму. Вторая часть сообщения шифруется публичным ключом предполагаемого получателя сообщения. В зашифрованной части содержится, так называемый, “nonce” – случайное число, генерируемое каждой стороной для очередного сеанса доверительного обмена. Сторона A генерирует свое случайное число, оно является ее частью их общего секрета. Сторона B генерирует в ответ свое число. В результате работы протокола каждая сторона должна быть уверена в своем партнере и должна узнать его часть секрета.

Полный вариант алгоритма построения доверительных отношений предполагает, что при построении отношений публичные ключи партнеров не хранятся на каждой стороне, за ними надо обращаться к серверу. Без потери общности будем считать, что каждый участник знает публичные ключи всех возможных партнеров. Сокращенный алгоритм Нидхама-Шредера таков:

Сообщение 1. Сторона A отправляет стороне B сообщение 1, в зашифрованной части которого передает свой идентификатор и nonceA. Сообщение шифруется открытым ключом B.

Сообщение 2. Сторона B, получив сообщение 1, расшифровывает его, проверяет соответствует идентификатор отправителя в зашифрованной части с идентификатором открытой части письма. Если соответствие есть, то сторона B уверена в своем партнере и отправляет сообщение 2, зашифровав публичным ключом A два секрета: nonceA и nonceB.

Сообщение 3. Сторона A, получив сообщение 2, расшифровывает его, находит в сообщении свой nonceA, что убеждает ее, что отправителем является B, поскольку только B мог прочитать зашифрованное сообщение 1. Теперь сторона A знает полный секрет, она подтверждает этот факт, посылая стороне B ее часть секрета nonceB в зашифрованной части сообщения 3. Сторона B, получив сообщение 3, уверена, что во-первых, ее партнер – A, во-вторых, A знает полный секрет.

Спустя почти 20 лет использования алгоритма, в 1995 году Лоуве (Lowe) описал модель алгоритма на CSP и верифицировал его при помощи верификатора FDR [4].

Он ввел в процесс обмена сообщениями третью сторону – злоумышленника (intruder), поведение которого не подчинялось определенном алгоритму. Злоумышленник может выполнять несколько действий с сообщениями:

1. прослушивать сообщения, идущие по каналу,
2. сохранять схваченные сообщения; если сообщения предназначались ему, то он расшифровывает скрытую часть, иначе, хранит в том виде, в котором ее получил,

3. генерировать сообщения на основе имеющейся информации,
4. отправлять свои сообщения в сеть.

Введения злоумышленника с заданным поведением оказалось достаточным для того, чтобы обнаружить криптографическую атаку в алгоритме Нидхама-Шредера.

Описание модели на языке Promela

Смоделируем протокол Нидхама-Шредера на языке Promela и попытаемся обнаружить атаку, найденную Лоуве, средствами Spin.

В модели присутствуют три участника `Alice` (сторона A), `Bob` (сторона B) и `Intruder` (злоумышленник), которым поставим в соответствие три одноименных процесса. Будем моделировать некорректное поведение алгоритма, поэтому считаем, что все сообщения в сети проходят через злоумышленника. Для обмена сообщениями между процессами введем три рандеву-канала: `fakedA` и `fakedB` (испорченные), `intercepted` (прослушиваемый). Записывая сообщение в `intercepted` канал, `Alice` или `Bob` не знают, что сообщение будет перехвачено. Прочитывая сообщение из своего испорченного канала `fakedA`, `Alice` не предполагает, что сообщение было испорчено (аналогичная ситуация для `Bob`). `Intruder` читает сообщение из канала `intercepted`, записывает сообщение в канал `fakedA` или в канал `fakedB` в зависимости от того, кому предназначается сообщение. Используется лишь один тип сообщения `msg`.

```
/* Читающий из faked канала не подозревает, что сообщение было испорчено */
chan fakedA = [0] of {mtype, mesCrypt};
chan fakedB = [0] of {mtype, mesCrypt};

/* Пишущий в intercepted канал не знает, что сообщение будет прочитано */
chan intercepted = [0] of {mtype, mesCrypt};
```

Структура сообщения сложная и состоит из нескольких частей, для того чтобы ее описать введены данные такого типа:

```
typedef mesCrypt {
       /* открытая часть сообщения */
            mtype  s,       /* отправитель */
             r,              /* получатель */
             nummsg,         /* номер сообщения */
             key,            /* публичный ключ, которым зашифровано сообщение */
      /* зашифрованная часть сообщения */
             d1,             /* здесь может быть идентификатор или nonce */
             d2
};
```

Кроме того, в перечислимых типах еще введены идентификаторы процесса, публичные ключи, случайные числа каждой стороны и состояния участников.

```
mtype  =  {msg,                                          /* тип сообщения */
          alice,  bob,  intruder,          /* идентификаторы участников */
          pkA, pkB, pkI,                   /* публичные ключи участников */
          nonceA,  nonceB,  nonceI,     /* случайные числа участников */
          ok, err};
```

Введем некоторые ограничения в модели для уменьшения числа возможных состояний системы:

- инициирует начало обмена сообщениями Alice,
- Intruder прослушивает сообщения от Alice и Bob, потому он знает их идентификаторы изначально,
- Intruder хранит лишь последнее прослушиваемое сообщение,
- получив сообщение, Intruder обязательно отправляет сообщение (прослушанное или сохраненное) в сеть,
- в случае если Alice или Bob получают сообщение, не соответствующее ожидаемому по любому из признаков (согласно действующему алгоритму), то процесс прекращает работу.

Получаемые из своих каналов данные, Alice и Bob хранят в переменной data описанного выше типа mesCrypt. Опишем лишь одну из проверок, которую проходит получаемое сообщение на честной стороне. Например, первое сообщение получает Bob

```
if  :: (data.r == bob) &&          /* Bob ли получатель сообщения ? */
           (data.key == pkB) &&      /* Можно ли расшифровать сообщение */
           (data.s == data.d1) &&    /* Можно ли доверять отправителю ? */
           (data.nummsg == 1)->      /* Правильный ли № сообщения ? */
           partnerB = data.s;
           pnonce = data.d2;
    :: else -> goto stopB;
fi;
```

Он проверяет, является ли он получателем сообщения, проверяет номер сообщения (должно прийти первое сообщение), проверяет, что информация зашифрована его ключом. Если все так, Bob может расшифровать сообщение. Затем Bob проверяет, что идентификатор отправителя в открытой части сообщения соответствует идентификатору в зашифрованной части сообщения. Он запоминает партнера и его часть секрета, только если все эти проверки пройдены, иначе он останавливает работу. Совершенно очевидно, что при таком количестве условий большинство попыток злоумышленника вторгнуться во взаимодействие Alice и Bob, закончатся безуспешно. Нас, однако, интересует другое: существуют ли успешные попытки вторжения. Оказывается, что такие возможности существуют, и они могут быть СГЕНЕРИРОВАНЫ системой верификации точно так же, как было сгенерировано решение в задаче “фермер, волк, коза, капуста“.

Intruder использует две переменные типа mesCrypt – переменную data для получения данных из канала, а другую fake для генерации своего сообщения на основе имеющейся информации. Злоумышленник не подвергает приходящие сообщения никаким проверкам, кроме одной: в случае, если зашифрованные данные зашифрованы его ключом (data.key == pkI), то он может расшифровать закрытую часть сообщения и узнать ее содержимое (значимые величины nonceA и nonceB). Остальные действия Intruder’a связаны с составлением своего сообщения, которое он отправит в сеть.

```
active proctype Intruder() {
           /* … */

end: do                                /* бесконечный цикл, в котором читаем
сообщения
                                       из канала intercepted */
     :: intercepted ? msg(data) ->
        if                             /* осуществляем проверку данных data */
        :: (data.key == pkI) ->        /* если можем расшифровать секретную часть,
```

```
                                        то извлекаем оттуда информацию в открытом
виде
                                        */
                 if
                 :: (data.d1 == nonceA || data.d2 == nonceA) -> knowNA =
true;          /* возможно нам попался nonceA, зафиксируем, что мы его знаем knowNA=1 */
                  /* аналогично для  nonceB */
                 /* … */
                 fi;
             :: else -> skip;           /* иначе: ничего не делаем */
             fi;

             /* построение испорченного сообщения fake */
             if                         /* недетерминировано выбираем отправителя */
             :: fake.s = alice ->       /* если отправитель Alice */
                fake.r = bob; fake.key = pkB;          /* то получатель Bob */
             /* аналогично для Bob и самого злоумышленника*/
              fi;

             /* поскольку злоумышленник не знает правила построения сообщения
             он помещает в часть сообщения d1 любую известную ему
             открытую информацию */
             if
             :: fake.d1 = alice;
             :: fake.d1 = bob;
             :: fake.d1 = intruder;
             :: fake.d1 = nonceI;
             :: (knowNA) -> fake.d1 = nonceA;
             :: (knowNB) -> fake.d1 = nonceB;
             fi;

             /* аналогично он поступает и с частью d2 */
             /* … */

             /* злоумышленник может вставить неизвестную зашифрованную часть из
             только что полученного сообщения */
             if
             :: (data.key != pkI) ->
                fake.key = data.key;
                fake.d1 = data.d1;
                fake.d2 = data.d2;
             :: else -> skip;
             fi;

            /* выбираем номер сообщения */
             /* … */

            /* отправляем сообщение или полученное, или свое созданное
            в канал, предназначая его либо Alice, либо Bob */
            if
            :: (data.r == alice) ->
                fakedA ! msg(data); /* повторяем полученное сообщение Alice*/
            :: (data.r == bob) ->
                fakedB ! msg(data); /* повторяем полученное сообщение Bob */
                /* аналогично для испорченного сообщения */
                /* … */
            fi;
```

```
    od;
}
```

Код алгоритма Нидхама-Шредера на языке Promela

```
/* Протокол Нидхама-Шредера */

/* Три участника: Alice, Bob, и Intruder, которые обмениваются сообщениями друг
   с другом по сети
*/

mtype = {msg, alice, bob, intruder, pkA, pkB, pkI,
      nonceA, nonceB, nonceI,  ok, err};

/* По сети передается сообщение
   ( отправитель, получатель, №, зашифрованные данные (случайные числа или ID) )
*/
typedef mesCrypt {                          /* сообщение */
             mtype  s, r, nummsg,           /* открытая часть */
                key, d1, d2;          /* зашифрованная часть */
};

/* Читающий из faked канала не подозревает, что сообщение было испорчено */
chan fakedA = [0] of {mtype, mesCrypt};
chan fakedB = [0] of {mtype, mesCrypt};

/* Пишущий в intercepted канал не знает, что сообщение будет прочитано */
chan intercepted = [0] of {mtype, mesCrypt};

/* Переменные, используемые в LTL формулах */
mtype partnerA, partnerB;
mtype statusA, statusB;

/* Переменные отражают знание посторонним секретных частей общего ключа */
bool knowNA, knowNB;

/* Честный инициатор процесса обмена */
active proctype Alice() {
     mtype  pkey, pnonce;
     mesCrypt  data;

     statusA = err;

      if /* недетерминировано выбирает партнера */
      :: partnerA = bob; pkey = pkB;
      :: partnerA = intruder; pkey = pkI;
      fi;

     /* конструируется сообщение № 1 и отправляется */
     d_step {
         data.s = alice;
         data.r = partnerA;
          data.nummsg = 1;
         data.key = pkey;
         data.d1  = alice;
         data.d2 = nonceA;
      }

     intercepted ! msg(data);

      /* ожидает сообщения  № 2 и расшифровывает его */
      fakedA ? msg(data);
```

```
end_errA:
      /* проверяет, что сообщение предназначалось именно ему и
            что отправитель его партнер,
            иначе останавливает процедуру обмена. */

      if
      :: (data.key == pkA) && (data.d1 == nonceA)  &&
         (data.s == partnerA) && (data.r == alice) &&
         (data.nummsg == 2)->
          pnonce = data.d2;
      :: else -> goto stopA;
      fi;

      /* отвечает сообщением №3 и успешно завершается*/
      d_step {
         data.s = alice;
         data.r = partnerA;
          data.nummsg = 3;
         data.key = pkey;
         data.d1 = pnonce;
         data.d2 = 0;
      }

      intercepted ! msg(data);
      statusA = ok;

stopA:
      printf("MSC: Process A finished \n");
} /* proctype Alice() */

 /* Честный партнер по обмену */
active proctype Bob() {
    mtype pkey, pnonce, receiver, partner;
    mesCrypt data;

    statusB = err;

    /* ожидает сообщения msg1, идентифицирует партнера */
    fakedB ? msg(data);

    /* проверяет сообщение на правильность построения,
       если что-то не соответствует ожиданию, останавливается */
end_errB1:
      if
      :: (data.r == bob) && (data.key == pkB) && (data.s == data.d1) &&
(data.nummsg == 1)->
            partnerB = data.s;
            pnonce = data.d2;
      :: else -> goto stopB;
      fi;

       /* устанавливает публичный ключ партнера */
       if
       :: (partnerB == alice) -> pkey = pkA;
       :: (partnerB == intruder) -> pkey = pkI;
       :: else -> goto stopB;
       fi;

        /* отвечает сообщением № 2 */
       d_step {
       data.s = bob;
       data.r = partnerB;
       data.nummsg = 2;
```

```
        data.key = pkey;
        data.d1 = pnonce;
        data.d2 = nonceB;
        }

       intercepted ! msg(data);

       /* ожидает № 3, проверяет сообщение на корректность,
              и в случае успеха завершается в состоянии ok */
       fakedB ? msg(data);

end_errB2:
       if
       :: (data.r == bob) && (data.s == partnerB) && (data.key == pkB) &&
(data.d1 == nonceB) && (data.nummsg == 3)->
            statusB = ok;
       :: else -> goto stopB;
       fi;

stopB:
       printf("MSC: Process B finished \n");
}

/* Злоумышленник не следует детерминированному алгоритму*/
active proctype Intruder() {
     mesCrypt data, fake;

     knowNA = false;
     knowNB = false;

end: do
     :: intercepted ? msg(data) ->
        if
        :: (data.key == pkI) ->
            if
            :: (data.d1 == nonceA || data.d2 == nonceA) -> knowNA = true;
            :: (data.d1 == nonceB || data.d2 == nonceB) -> knowNB = true;
            fi;
        :: else -> skip;
        fi;

        /* построение испорченного сообщения */
        if
        :: fake.s = alice ->
            fake.r = bob; fake.key = pkB;
        :: fake.s = bob;
            fake.r = alice; fake.key = pkA;
        :: fake.s = intruder;
            if
            :: fake.r = bob; fake.key = pkB;
            :: fake.r = alice; fake.key = pkA;
            fi;
        fi;

        /* злоумышленник не знает правила построения сообщения */
        if
        :: fake.d1 = alice;
        :: fake.d1 = bob;
        :: fake.d1 = intruder;
        :: fake.d1 = nonceI;
        :: (knowNA) -> fake.d1 = nonceA;
        :: (knowNB) -> fake.d1 = nonceB;
        fi;
```

```
            if
            :: fake.d2 = alice;
            :: fake.d2 = bob;
            :: fake.d2 = intruder;
            :: fake.d1 = nonceI;
            :: (knowNA) -> fake.d2 = nonceA;
            :: (knowNB) -> fake.d2 = nonceB;
            fi;

            /* вставляет в сообщение неизвестную зашифрованную часть
                  из полученного сообщения */
            if
            :: (data.key != pkI) ->
                 fake.key = data.key;
                 fake.d1 = data.d1;
                 fake.d2 = data.d2;
            :: else -> skip;
            fi;

            /* выбираем номер сообщения */
           if
           :: fake.nummsg = 1;
           :: fake.nummsg = 2;
           :: fake.nummsg = 3;
           fi;

            if
            :: (data.r == alice) ->
                 fakedA ! msg(data); /* повторяем полученное сообщение*/
            :: (data.r == bob) ->
                 fakedB ! msg(data); /* повторяем полученное сообщение*/
            :: (fake.r == alice)
                 fakedA ! msg(fake); /* отправляем испорченное сообщение */
            :: (fake.r == bob)
                 fakedB ! msg(fake); /* отправляем испорченное сообщение */
            fi;
      od;
   }
```

Поиск криптографической атаки

В описанном алгоритме атакой можно считать ситуацию, когда сторона B будет считать своим партнером сторону A, сторона A будет доверять своему партнеру, а злоумышленник будет знать обе части их секрета. Сформулируем эту ситуацию в виде LTL формулы: "не существует такого пути в системе, что когда-нибудь в будущем сторона B, успешно завершив работу, будет полагать, что ее партнер A, и подслушивающий при этом знает `nonceA` и `nonceB`"

```
<> ( agentB_finished && bobtrustsA && intrknowNA && intrknowNB )
```

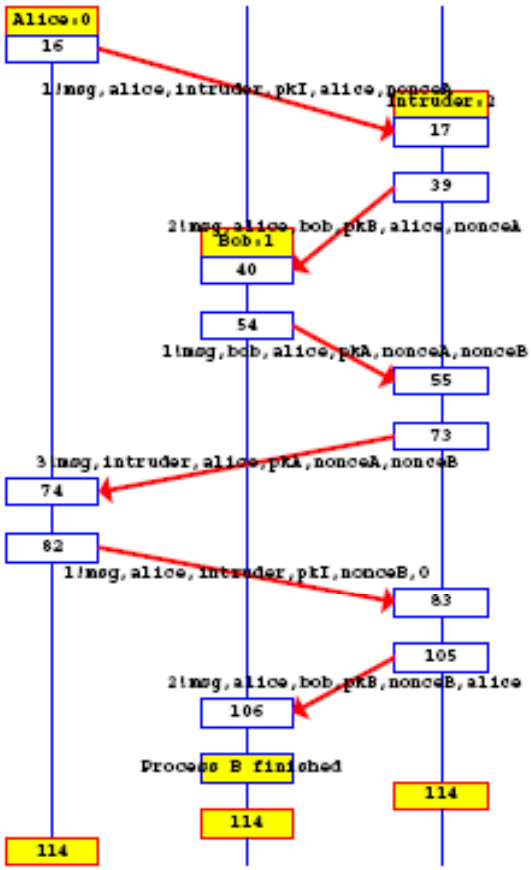
где предикаты заданы следующим образом (переменные `statusB`, `partnerB`, `knowNA`, `knowNB` определены в теле программы)

```
#define agentB_finished (statusB == ok)
#define bobtrustsA            (partnerB == alice)
#define intrknowNA            (knowNA == true)
#define intrknowNB            (knowNB == true)
```

Поставив флаг *No Executions* для заданной формулы, запустим верификатор Spin. Spin обнаружил атаку, описанную Лоуве в 1995 году.

1. Alice, считая Intruder честной стороной, посылает ему сообщение № 1 со своим идентификатором и своей частью секретного ключа, зашифровав все публичным ключом Intruder [16 -> 17]. Intruder уже известен nonceA.

2. Intruder посылает сообщение № 1 Bob, поставив в открытую часть сообщения в качестве отправителя Alice, а далее повторив ее письмо себе, зашифровав секретную часть публичным ключом Bob [39-40].



3. Bob отвечает Intruder сообщением № 2, полагая, что тот является Alice. В письме содержится часть, непонятная Intruder, поскольку она зашифрована публичным ключом Alice [54-55]. В зашифрованной части письма идет nonceB – часть секрета Bob, но она не доступна злоумышленнику.

4. Злоумышленник отправляет сообщение № 2 Alice [73-74]. В качестве основы он берет сообщение Bob, зашифрованную часть он вставляет целиком, а в открытой части меняет отправителя на себя.

5. Alice, доверяя Intruder, расшифровывает сообщение № 2, извлекает nonceB, отвечает Intruder сообщением № 3, содержащим nonceB, при чем секретная часть зашифрована публичным ключом Intruder [82-83].

6. Intruder расшифровывает сообщение № 3, узнает nonceB, посылает сообщение № 3 Bob. Полученное сообщение проходит все проверки Bob, и он успешно завершает работу [105-106].

3.4 Задача об обедающих философах

Описание алгоритма

Знаменитая задача об обедающих философах, предложенная Эдсгером Дейкстрой, является образным описанием – метафорой, позволяющей ясно представить проблемы, возникающие при совместном использовании ресурсов несколькими параллельными процессами.

За круглым столом сидят несколько философов, каждый из которых проводит время в размышлениях, изредка прерываясь на еду, когда он проголодается. На столе стоит блюдо со спагетти и вилки, по одной между каждыми двумя философами. Для того чтобы поесть, каждый философ должен использовать две вилки, которые лежат непосредственно слева и справа от него. Проголодавшись, он по очереди берет вилки, если они не заняты, ест спагетти, после чего возвращает вилки на место и продолжает думать до тех пор, пока снова не проголодается, затем опять ест и т.д. Если правая либо левая вилка занята соседом, философ просто ждет ее освобождения.

Сформулированная таким образом ситуация кажется совершенно безобидной, поведение каждого процесса очевидно, и трудно предположить, что в этой просто сформулированной ситуации могут возникнуть проблемы. В то же время формальное представление ситуации в виде параллельных процессов, когда философы представляют процессы, а вилки представляют общие ресурсы, демонстрирует возможность возникновения блокировки процессов, конкурирующих за разделяемые ресурсы. Такая блокировка возникает очень редко, но она приводит к остановке всех процессов. На языке метафоры это означает, что может сложиться условие, при которой все философы, упрямо следующие своим правилам поведения (фактически, просто выполняющие свой алгоритм поведения), умрут голодной смертью даже при наличии полного блюда спагетти.

Построим формальное описание ситуации. Опишем возможные состояния философа:

S0. Философ размышляет. В этом состоянии он может проголодаться, что представляется спонтанным переходом в состояние S1.

S1. В этом состоянии философ хочет есть, поэтому ему требуются вилки. Философ тянется за вилкой, лежащей слева от него. Если эта вилка отсутствует, то он в состоянии S1 ждет ее освобождения. Если левая вилка свободна, то он ее берет и переходит в состояние S2.

S2. В этом состоянии философ ожидает правую вилку. Если правая вилка отсутствует, то он будет находиться в этом состоянии до ее освобождения. Если правая вилка свободна, то он ее берет и переходит в состояние S3 – имея обе вилки, он в этом состоянии ест спагетти.

S3. Философ ест. После того, как он поел, он кладет на стол левую вилку, переходя в следующее состояние S4.

S4. В этом состоянии философ кладет на стол правую вилку и возвращается к размышлениям (в состояние S0).

Возможные состояния каждой вилки:

F0. Вилка свободна. Она лежит на столе и готова к тому, что ее возьмет один из философов (слева или справа). Если ее берет левый философ, она переходит в состояние F1, если ее берет правый философ, она переходит в состояние F2.

F1. Вилка находится у философа слева и ожидает возвращения на стол (состояние F0).

F2. Вилка находится у философа справа и ожидает возвращения на стол (состояние F0).

Возможна ли ситуация общего голодания, т.е. когда все философы не смогут поесть из-за отсутствия у них необходимых вилок?

Мы обнаружим ситуацию общего голодания при помощи пакета верификации параллельных процессов SPIN.

Описание модели на языке Promela

Решаем задачу для N философов. Введем в модель 2N процессов: N процессов потребуется для описания поведения философов и N процессов – для описания поведения вилок.

Состояние философа фиксируем в переменной `cur_state`. Взятию вилки философом и ее возвращению (освобождению) сопоставим передачу сообщения по рандеву-каналу от философа к вилке. Всего вводится 2N каналов.

```
chan l_fork[N] = [0] of {bit}; /* Для левой к философу вилки */
chan r_fork[N] = [0] of {bit}; /* Для правой к философу вилки */
```

Каждому философу необходимо два канала – один для организации обмена с левой вилкой, другой – для организации обмена с правой вилкой. Использование рандеву-каналов

гарантирует синхронизацию взаимодействия между философом и вилкой: философ получит вилку только тогда, когда вилка будет свободна; лежащая на столе вилка поступит в распоряжение философа только тогда, когда философ ее запросит.

Для передачи запроса о доступе к вилке и об ее освобождении будет достаточно сообщений одного типа:

```
#define msgtype 1     /* тип сообщения */
```

которые фактически будут играть роль сигнала для вилки и философа для изменения состояния.

```
proctype phil (chan left, right; byte mn)
{
      byte  cur_state = 0;

                  printf("MSC: phil # %d \n", mn);

      do
      ::  (cur_state == 1)  ->
            left ! msgtype  ->      /* Философ запросил левую вилку */
            cur_state = 2;

      ::  (cur_state == 2)  ->
            right ! msgtype;        /* Философ запросил правую вилку */
            cur_state = 3;          /* Философ ест */

      ::(cur_state == 3) ->
             cur_state = 4;         /* Философ наелся */

      :: (cur_state == 4) ->
            right ! msgtype ->      /* Философ вернул правую вилку */
            cur_state = 5;

      :: (cur_state == 5) ->
            left ! msgtype ->       /* Философ вернул левую вилку */
            cur_state = 0;          /* Философ размышляет */
      ::(cur_state == 0) ->
            cur_state = 1;          /* Философ решил поесть */

      od
}
```

Для хранения состояния процесса-вилки используется переменная `cur_state_fork`. Взаимодействие с философами осуществляется по тем же рандеву-каналам – по одному каналу на каждого философа (один канал для левого философа, один для правого). Когда вилка лежит на столе (она свободна), то ожидается запрос либо от левого, либо от правого философа на ее использование. Когда один из философов взял вилку, то по соответствующему каналу ожидается сигнал о возвращении вилки.

```
proctype fork(chan left_phil, right_phil)
{     byte  cur_state_fork = 0;           /* F0. Вилка свободна */

end:  do
      ::( cur_state_fork == 0) ->         /* Если вилка свободна */
            if
            ::right_phil ? msgtype ->     /* Правый философ запросил вилку */
                  cur_state_fork = 2;     /* F2. Вилка у правого философа */

            ::left_phil ? msgtype ->      /* Левый философ запросил вилку */
                  cur_state_fork = 1;     /* F1. Вилка у левого философа */
```

```
            ::skip;

            fi
      ::( cur_state_fork == 1) ->
               left_phil ? msgtype ->     /* Левый философ возвращает вилку */
                   cur_state_fork = 0;    /* F0. Вилка свободна */

      ::( cur_state_fork == 2) ->
               right_phil ? msgtype ->    /* Правый философ возвращает вилку */
                   cur_state_fork = 0;    /* F0. Вилка свободна */
      od
}
```

Запуск процессов-вилок и процессов-философов происходит в процессе `init`. Рассмотрим нумерацию процессов и каналов. Даны `N` процессов-философов с номерами от `1` до `N` (аналогично для вилок), `N` каналов для левых (аналогично для правых) по отношению к философам вилок с номерами от `0` до `N-1`.

- Припишем процессу-философу номер `proc`.
- `l_fork` – канал для взаимодействия философа `proc` с левой вилкой получит номер `proc-1`.
- `r_fork` – каналу для взаимодействия философа `proc` с правой вилкой назначим номер `proc`.
- Процессу, соответствующему правой вилке этого философа, назначим номер `proc`.

Согласно предложенной нумерации получим номера и названия каналов для взаимодействия вилки с левым и правым философом: `r_fork` с номером `proc` для левого по отношению к вилке философа и `l_fork` с номером `proc` для правого по отношению к вилке философа. Запуск начинается с процессов с номером `1`.

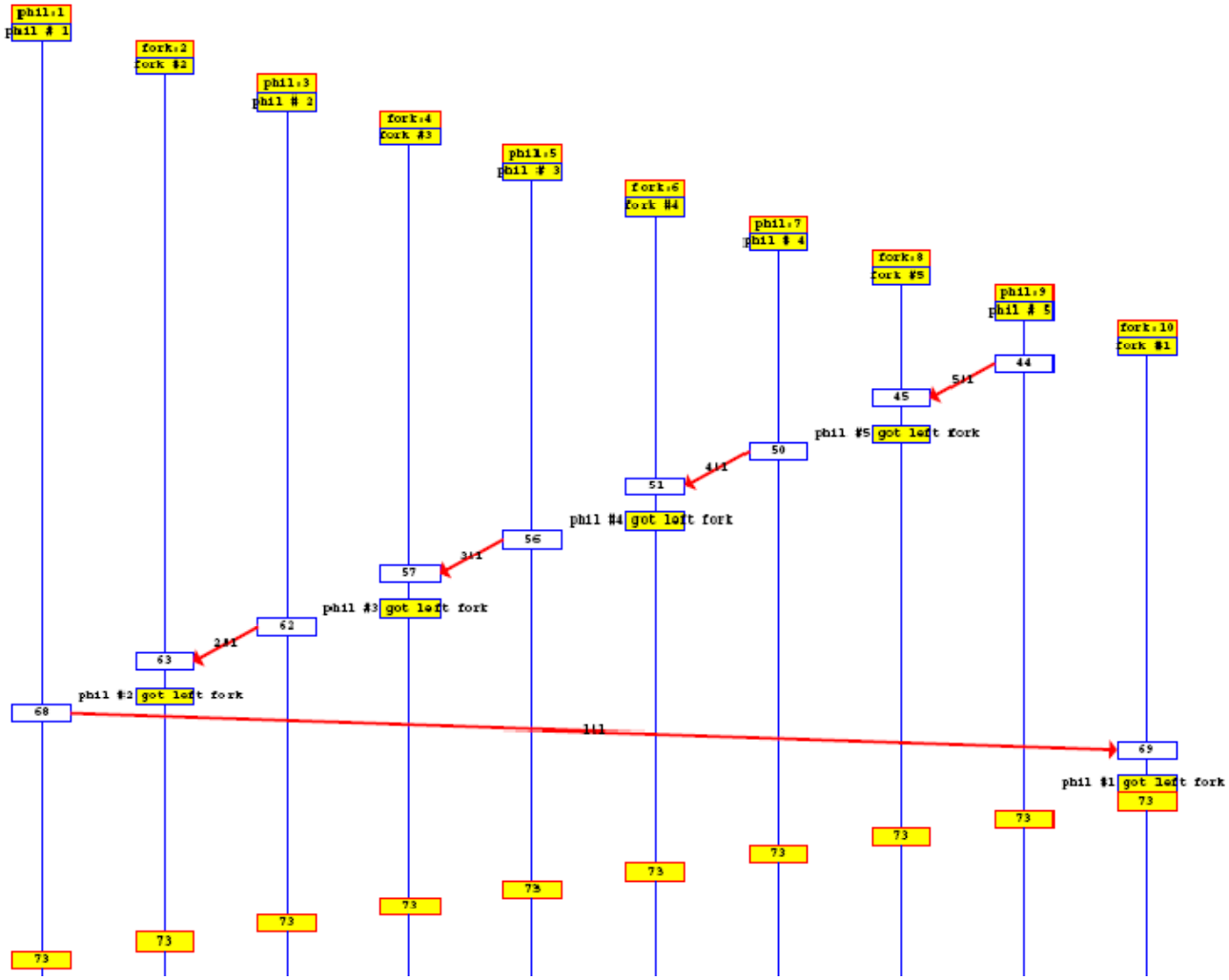
```
init {
      byte  proc;

      proc = 1;
      get_in_stuck = 0;

      do
      :: proc <= N  ->

            /* запуск процесса-философа №proc с левым каналом №(proc-1) и правым каналом №proc */
            run phil (l_fork[proc-1], r_fork[proc%N], proc);
            /* запуск процесса-вилки №proc с каналом для левого философа №proси каналом для левого
философа №proc */
            run fork (r_fork[proc%N], l_fork[proc%N], proc);

            proc++;

      :: proc > N ->
            break
      od
}
```

Обнаружение состояния общего голодания

Ситуация общего голодания, которую необходимо обнаружить, проявляется во взаимной блокировке процессов – ни один из процессов-философов не может продолжить работу, ожидая поступления вилки. Свойство взаимной или системной блокировки (system deadlock)

относится к базовым верифицируемым Spin и проверяется автоматически при поиске некорректных конечных состояний. Для этого необходимо убедиться, что в окне *Basic Verification Options* установлен флаг *Invalid Endstates*. В выводе результатов верификации появится сообщение, свидетельствующее об обнаружении пакетом системной блокировки:

```
Full statespace search for:
      never claim             - (not selected)
      assertion violations    - (disabled by -A flag)
      cycle checks            - (disabled by -DSAFETY)
      invalid end states      +
```



Контрпример, обнаруженный верификатором, демонстрирует, как один за другим философы тянутся за левыми вилками, до тех пор пока не приходят к ситуации, когда каждый философ ждет правой вилки.

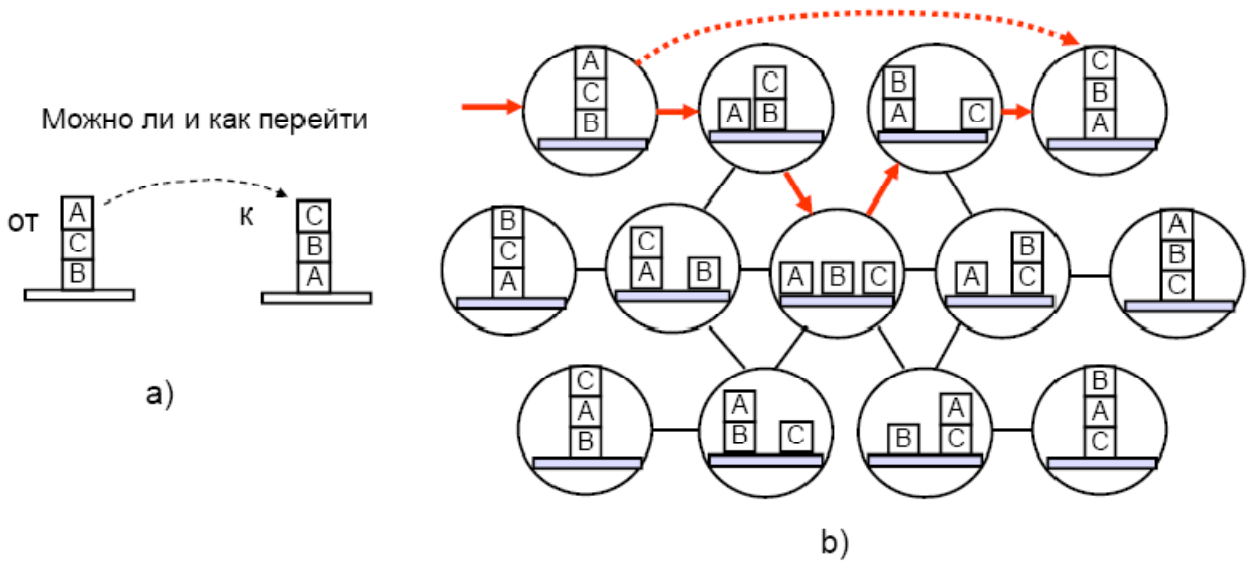
3.5 Мир блоков

Мир блоков – это искусственная среда планирования действий, которая вследствие своей ясности и простоты была одним из наиболее часто приводимых в литературе по планированию в искусственном интеллекте в 1960х.

Среда состоит из кубиков (блоков), стоящих на столе по отдельности или в нескольких вертикальных башнях. Взаимное расположение кубиков можно считать состоянием среды. Целью является построение заданной конфигурации – одной или нескольких башен. Только один кубик, на котором ничего не стояло, может быть перемещен за один шаг: он может быть либо поставлен на стол, либо помещен сверху на какую-нибудь башню.

Описание модели на языке Promela

Построим модель на языке Promela, описывающую все возможные конфигурации и допустимые их изменения для любого заданного числа блоков и любого их расположения. Число блоков определим параметром `N`, блок будет идентифицироваться уникальным номером от `0` до `N-1`. Конфигурацию будем задавать двумя массивами длины `N`: `up[i]` определяет номер блока, стоящего над `i-м`, `down[i]` определяет номер блока, стоящего под `i-м`. Специальная константа `NOTHING` определяет, что над или под блоком ничего не стоит. Целевая конфигурация определяется подобными массивами `Tup` и `Tdown`.



Исходная конфигурация на рисунке будет представлена так: `up[0]=NOTHING, up[1]=2, up[2]=0, down[0]=2, down[1]=NOTHING, down[2]=2`. Возможные переходы между состояниями программируются так:

a) выбирается случайный блок `i`, стоящий сверху (над ним ничего нет, `up[i]==NOTHING`), и этот блок снимается (`up[down[i]] = NOTHING`);

b) выбранный блок либо кладется на стол (`down[i]=NOTHING`), либо выбирается другой блок `j`, который стоит наверху (`up[j]==NOTHING`), и на него помещается блок `i` (`up[j]=i; down[i] = j`). Блок `j` отличен от `i` (`j≠i`),.

Проверяемое условие задается как совпадение текущей и целевой конфигурации, т.е. совпадением массивов `up[]` и `Tup[]`, а также `down[]` и `Tdown[]`. Программа в цикле сначала проверяет, совпали ли текущая и целевая конфигурации, и если нет – то случайно делает один из возможных переходов – изменяет текущее состояние (конфигурацию) на соседнее.

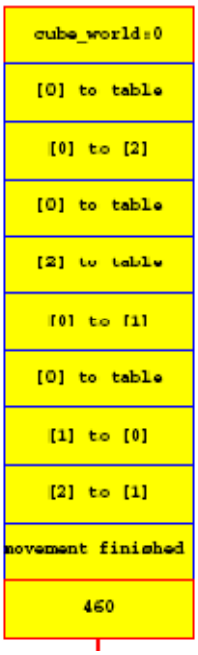
```
#define N 3                          /* число кубиков */
#define NOTHING 255
#define N-1 2

/* массивы текущего состояния */
byte  up    [N];              /* номер над i -м кубиком */
byte down[N];                 /* номер под i -м кубиком */

/* массивы целевого состояния */
byte Tup     [N];          /* номер над i -м кубиком */
byte Tdown[N];             /* номер под i -м кубиком */

byte i, j, k;                          /* номера кубиков */
bool equal = false,
         findi = true,                 /* определяем снимаемый кубик i */
         findj = false,          /* определяем кубик j, на который ставим i */
```

```
          checkconf = false;      /* проверяем полученную конфигурацию */

active proctype cube_world(){
  /* начальное состояние*/
  up[0] = NOTHING;
  up[1] = 2;
  up[2] = 0;
  down[0] = 2;
  down[1] = NOTHING;
  down[2] = 1;

 /* целевое состояние */
  Tup[0] = 1;
  Tup[1] = 2;
  Tup[2] = NOTHING;
  Tdown[0] = NOTHING;
  Tdown[1] = 0;
  Tdown[2] = 1;

    i = 0;
  j = 0;
  /* основной цикл */
  do
       :: findi ->    /* случайный выбор  i  */
            if  :: (up[i] == NOTHING) ->  findj = true; findi = false;
                      if  :: ( down[i] != NOTHING ) ->
                                up[ down[i] ] = NOTHING;     /* i-й сняли */
                             :: else -> skip;
                      fi;
                    :: (i < N-1) -> i ++
                    :: (i > 0)   -> i --
            fi

       :: findj && ( down[i] != NOTHING) -> /* */
          down[i] = NOTHING;           /* i-й положили на стол */
          printf("MSC:  [%d] to table \n",  i);
          findj = false;
          checkconf = true;

       :: findj ->  /* случайно выберем  j  и  i-й поставим на  j-й */
          if  :: ( i != j ) && ( up[j] == NOTHING ) ->
                    up[j] = i; down[i] = j;      /* i-й кубик ставим на  j-й */
                  printf("MSC:  [%d] to [%d] \n",  i, j);
                  findj = false;
                  checkconf = true;
              :: (j < N-1) -> j ++
              :: (j > 0)     -> j --
          fi

       :: checkconf -> /* проверка, достигли ли целевого состояния */
          equal = true; k = 0;
          do  :: ( k < N ) ->
                  equal=equal && (up[k]==Tup[k]) && (down[k]==Tdown[k]);
                  k++
              :: ( !equal )  -> break
              :: ( k == N ) -> break
          od;
          if  :: (equal) -> break /* целевое состояние найдено */
              :: else -> checkconf = false; findi = true
          fi

  od;
```

```
            printf("MSC: movement finished \n")
    }
```

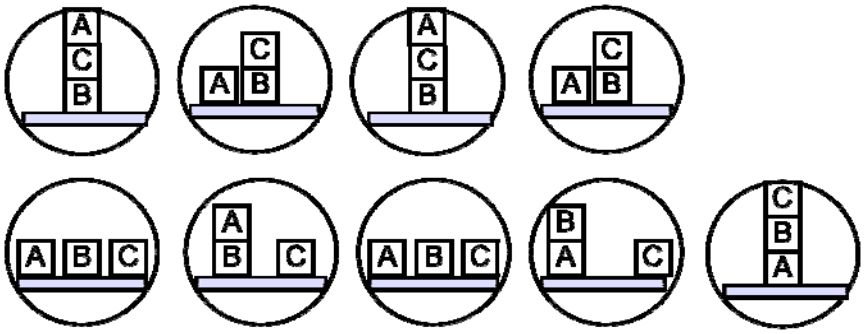
Поиск решения задачи средствами Spin

Задав начальное и целевое состояния, попробуем найти решение (последовательность допустимых переходов) средствами верификации. Для этого поставим задачу перед Spin: не существует ни одного пути такого, что выполняется LTL-формула: `FG (equal == true)`, т.е. не существует ни одного пути, на котором возможно прийти к целевому состоянию из данного начального.



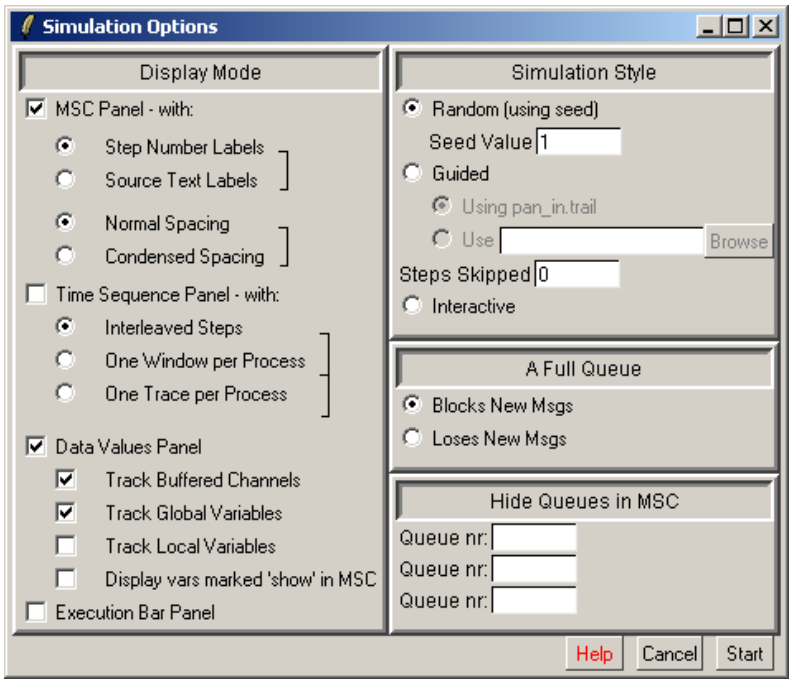
Spin находит контрпример, предлагая решение данной задачи.

Поскольку модель допускает несколько решений, то Spin находит одну из возможных последовательностей переходов – совсем неоптимальную. Изобразим для наглядности решение, предложенное верификатором, как последовательность перемещения кубиков.



Приложение 1. Параметры симуляции

Выберите в меню *Run* пункт *Set Simulation Parameters*. Откроется диалоговое окно *Simulation Options.*



В данном окне на панели *Display Mode* задается, какие окна необходимо выводить при симуляции.

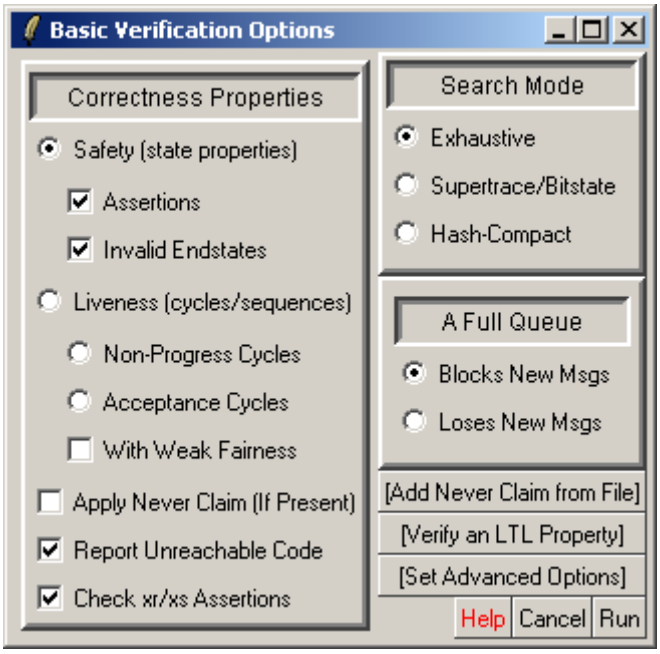
- *MSC Panel* – панель диаграммы взаимодействия. Для данного окна можно задать, как будут отображаться надписи на диаграмме.
    - *Step Number Labels* – на диаграмме указываются номера шагов симуляции.
    - *Source Text Labels* – на диаграмме указываются строчки исходного кода.*Ghj*
- *Time Sequence Panel*. Данная панель показывает, какие действия выполняет каждый процесс в определенный момент времени.
    - *Interleaved Steps* – в каждый момент времени на диаграмме выводится номер процесса, который выполняется, и строка кода, которая выполняется в этом процессе.
    - *One Window per Process* – для каждого процесса открывается окно, содержащее описание процесса на языке Promela. Во время симуляции подсвечивается строка, которая в данный момент времени выполняется.
    - *One Trace per Process* – для каждого процесса открывается окно. Во время симуляции выводится номер строки, которая в данный момент времени выполняется.
- *Data Values Panel* – панель для просмотра значений данных.
    - *Track Buffered Channels* – просмотр значений каналов с буфером.
    - *Track Global Variables* – просмотр значения глобальных переменных.
    - *Track Local Variables* – просмотр значения локальных переменных.
    - *Display vars marked ‘show’ in MSC* – просмотр изменения значения переменной на MSC диаграмме. Для отображения переменной необходимо перед

объявлением этой переменной поставить префикс `show` (например, написать `show byte cnt` вместо `byte cnt`).

- *Execution Bar Panel* – панель просмотра процента выполненных шагов каждым процессом в зависимости от общего числа выполненных шагов.
- На панеле *Simulation Style* устанавливается тип симуляции. Возможны 3 вида симуляции:
  - *Random* - все недетерминированные решения определяются случайным образом.
  - *Guided* – симуляция для сгенерированного верификатором ошибочного пути. Данная симуляция используется для отладки ошибки, найденной верификатором.
  - *Steps Skipped* – число первых шагов, которые пропускаются.
  - *Interactive* – все недетерминированные решения запрашиваются у пользователя.

Приложение 2. Параметры верификации базовых свойств

Выберете пункт *Set Verification Parameters* меню *Run*. Откроется диалоговое окно *Basic Verification Options.*



На панели *Correctness Properties* указываются базовые свойства, которые можно верифицировать:

- *Safety* – свойства безопасности
  - *Assertions* – проверка сохранения локальных инвариантов, описанных при помощи оператора `assert` (см. 2.1);
  - *Invalid Endstates* – проверка на наличие некорректных конечных состояний, т.е. обнаружение взаимных блокировок процессов (см. 2.2);
- *Liveness* – свойства живости
  - *Non-Progress Cycles* – проверка отсутствия бесконечных циклов, не содержащих операторов, помеченных меткой `progress` (см. 2.3);
  - *Acceptance Cycles* – проверка отсутствия циклов, с бесконечно частым выполением операторов с меткой `accept` (см. 2.4).
- *Apply Never Claim* – учитывать процесс never, если таковой имеется в тексте модели (см. 2.5).
- *Report Unreachable Code* – выводить сообщения о недостижимом коде.
- *Check xr/xs Assertions (channel assertions)* – проверять, что только процесс, в котором описан канал с использованием оператора `xr`, имеет право на чтение данных из канала. Аналогично, процесс, в котором канал описан с использованием `xs` оператора, имеет эксклюзивное право на запись данных в канал.

Литература

1. Holzmann G. "Spin Model Checker. The Primer and Reference Manual" Addison Wesley, 2003, 608 стр.
2. Peterson G. "An *O(n* log n) Unidirectional Algorithmfor the Circular Extrema Problem" ACM Transactions on Programming Languages and Systems (TOPLAS), v. 4 , i. 4, p. 758 – 762, 1982
3. Dolev D., Klawe M., Rodeh M. "An O(n log n) Unidirectional Distributed Algorithm for Extrema Finding in a Circle" Journal of Algorithms, 3, стр. 245-260, 1982
4. Lowe G. " Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR" Lecture Notes In Computer Science; v. 1055, Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems table of contents, стр. 147 – 166, 1996
5. Карпов Ю.Г.. Model checking. Верификация параллельных и распределенных программных систем. // БХВ-Петербург, 2009, 520 с.