Автоматное программирование

Хлебников Андрей Александрович

21 марта 2018 г.

Оглавление

Введение

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

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

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

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

Парадигмы программирования

Императивное программирование

Императивное программирование – это парадигма программирования (стиль написания исходного кода компьютерной программы), для которой характерно следующее:

- в исходном коде программы записываются инструкции (команды);
- инструкции должны выполняться последовательно;
- при выполнении инструкции данные, полученные при выполнении предыдущих инструкций, могут читаться из памяти;
- данные, полученные при выполнении инструкции, могут записываться в память.

Императивная программа похожа на приказы (англ. **imperative** – приказ, повелительное наклонение), выражаемые повелительным наклонением в естественных языках, то есть представляют собой последовательность команд, которые должен выполнить компьютер.

При императивном подходе к составлению кода (в отличие от функционального подхода, относящегося к декларативной парадигме) широко используется присваивание. Наличие операторов присваивания увеличивает сложность модели вычислений и делает императивные программы подверженными специфическим ошибкам, не встречающимся при функциональном подходе[1].

Основные черты императивных языков:

- использование именованных переменных;
- использование оператора присваивания;
- использование составных выражений;
- использование подпрограмм;

[1]Harold Abelson, Jerry Sussman, and Julie Sussman: Structure and Interpretation of Computer Programs (MIT Press, 1984; ISBN 0-262-01077-1), Pitfalls of imperative programming

История: Первыми императивными языками были машинные инструкции (коды) - команды, готовые к исполнению компьютером сразу (без каких-либо преобразований). В дальнейшем были созданы ассемблеры, и программы стали записывать на языках ассемблеров. Ассемблер - компьютерная программа, предназначенная для преобразования машинных инструкций, записанных в виде текста на языке, понятном человеку (языке ассемблера), в машинные инструкции в виде, понятном компьютеру (машинный код). Одной инструкции на языке ассемблера соответствовала одна инструкция на машинном языке. Разные компьютеры поддерживали разные наборы инструкций. Программы, записанные для одного компьютера, приходилось заново переписывать для переноса на другой компьютер. Были созданы языки программирования высокого уровня и компиляторы - программы, преобразующие текст с языка программирования на язык машины (машинный код). Одна инструкция языка высокого уровня соответствовала одной или нескольким инструкциям языка машины, и для разных машин эти инструкции были разными. Первым распространённым высокоуровневым языком программирования, получившим применения на практике, стал язык Fortran

Декларативное программирование

Декларативное программирование – это парадигма программирования, в которой задаётся спецификация решения задачи, то есть описывается, что представляет собой проблема и ожидаемый результат. Противоположностью декларативного является императивное программирование, описывающее на том или ином уровне детализации, как решить задачу и представить результат. В общем и целом, декларативное программирование идёт от человека к машине, тогда как императивное - от машины к человеку. Как следствие, декларативные программы не используют понятия состояния, то есть не содержат переменных и операторов присваивания (см. также ссылочная прозрачность).

Наиболее близким к «чисто декларативному» программированию является написание исполнимых спецификаций (см. соответствие Карри - Ховарда). В этом случае программа представляет собой формальную теорию, а её выполнение является одновременно автоматическим доказательством этой теории, и характерные для императивного программирования составляющие процесса разработки (проектирование, рефакторинг, отладка и др.) в этом случае исключаются: программа проектирует и доказывает сама себя.

К подвидам декларативного программирования также зачастую относят функциональное и логическое программирование - несмотря на то, что программы на таких языках нередко содержат алгоритмические составляющие, архитектура в императивном понимании (как нечто отдельное от кодирования) в них также отсутствует: схема программы является непосредственно частью исполняемого кода(http://fprog.ru/2010/issue6/interview-simon-peyton-jones/).

На повышение уровня декларативности нацелено языково-ориентированное программирование.

«Чисто декларативные» компьютерные языки зачастую не полны по Тьюрингу - примерами служат SQL и HTML - так как теоретически не всегда возможно

порождение исполняемого кода по декларативному описанию. Это иногда приводит к спорам о корректности термина «декларативное программирование» (менее спорным является «декларативное описание решения» или, что то же самое, «декларативное описание задачи»).

Структурное программирование

Структурное программирование – методология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры блоков. Предложена в 1970-х годах Э. Дейкстрой и др.

В соответствии с данной методологией любая программа строится без использования оператора goto из трёх базовых управляющих структур: последовательность, ветвление, цикл; кроме того, используются подпрограммы. При этом разработка программы ведётся пошагово, методом «сверху вниз».

Методология структурного программирования появилась как следствие возрастания сложности решаемых на компьютерах задач, и соответственно, усложнения программного обеспечения. В 1970-е годы объёмы и сложность программ достигли такого уровня, что традиционная (неструктурированная) разработка программ перестала удовлетворять потребностям практики. Программы становились слишком сложными, чтобы их можно было нормально сопровождать. Поэтому потребовалась систематизация процесса разработки и структуры программ.

Методология структурной разработки программного обеспечения была признана «самой сильной формализацией 70-х годов».

По мнению Бертрана Мейера, «Революция во взглядах на программирование, начатая Дейкстрой, привела к движению, известному как структурное программирование, которое предложило систематический, рациональный подход к конструированию программ. Структурное программирование стало основой всего, что сделано в методологии программирования, включая и объектное программирование»[2]

Цель структурного программирования - повысить производительность труда программистов, в том числе при разработке больших и сложных программных комплексов, сократить число ошибок, упростить отладку, модификацию и сопровождение программного обеспечения.

Функциональное программирование

Функциональное программирование – раздел дискретной математики и парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании).

Противопоставляется парадигме императивного программирования, которая описывает процесс вычислений как последовательное изменение состояний (в значении, подобном таковому в теории автоматов). При необходимости,

[2]Мейер Б. Почувствуй класс. Учимся программировать хорошо с объектами и контрактами. - Пер. с англ. - М.: Национальный открытый университет ИНТУИТ: БИНОМ. Лаборатория знаний, 2011. - 775с. - С. 208. - ISBN 978-5-9963-0573-5

в функциональном программировании вся совокупность последовательных состояний вычислительного процесса представляется явным образом, например, как список.

Функциональное программирование предполагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Соответственно, не предполагает оно и изменяемость этого состояния (в отличие от императивного, где одной из базовых концепций является переменная, хранящая своё значение и позволяющая менять его по мере выполнения алгоритма).

На практике отличие математической функции от понятия «функции» в императивном программировании заключается в том, что императивные функции могут опираться не только на аргументы, но и на состояние внешних по отношению к функции переменных, а также иметь побочные эффекты и менять состояние внешних переменных. Таким образом, в императивном программировании при вызове одной и той же функции с одинаковыми параметрами, но на разных этапах выполнения алгоритма, можно получить разные данные на выходе из-за влияния на функцию состояния переменных. А в функциональном языке при вызове функции с одними и теми же аргументами мы всегда получим одинаковый результат: выходные данные зависят только от входных. Это позволяет средам выполнения программ на функциональных языках кешировать результаты функций и вызывать их в порядке, не определяемом алгоритмом и распараллеливать их без каких-либо дополнительных действий со стороны программиста (что обеспечивают функции без побочных эффектов - чистые функции).

Лямбда-исчисление являются основой для функционального программирования, многие функциональные языки можно рассматривать как «надстройку» над ними[3].

Пример(Erlang):

```
proc(Function, List, Number) ->
    process_flag(trap_exit, true),
    Supervisor = self(),
    spawn_link(combinat, Function, [List, Number, fun(R)->Supervisor!R end]),
    loop([]).

loop(Total) ->
    receive
    'EXIT', Worker, normal ->
        io:format("~w~n", [Total]),
        unlink(Worker);
    Result ->
        loop(Total ++ [Result])
    end.
```

[3]А. Филд, П. Харрисон Функциональное программирование: Пер. с англ. - М.: Мир, 1993. - 637 с, ил. ISBN 5-03-001870-0. Стр. 120 [Глава 6: Математические основы: Лямбда-исчисление]

Логическое программирование

Логическое программирование – парадигма программирования, основанная на автоматическом доказательстве теорем, а также раздел дискретной математики, изучающий принципы логического вывода информации на основе заданных фактов и правил вывода. Логическое программирование основано на теории и аппарате математической логики с использованием математических принципов резолюций.

Самым известным языком логического программирования является Prolog.

Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) – методология программирования основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования[4].

Идеологически ООП - подход к программированию как к моделированию информационных объектов, решающий на новом уровне основную задачу структурного программирования: структурирование информации с точки зрения управляемости[5], что существенно улучшает управляемость самим процессом моделирования, что в свою очередь особенно важно при реализации крупных проектов.

Управляемость для иерархических систем предполагает минимизацию избыточности данных (аналогичную нормализации) и их целостность, поэтому созданное удобно управляемым - будет и удобно пониматься. Таким образом через тактическую задачу управляемости решается стратегическая задача - транслировать понимание задачи программистом в наиболее удобную для дальнейшего использования форму.

Основные принципы структурирования в случае ООП связаны с различными аспектами базового понимания предметной задачи, которое требуется для оптимального управления соответствующей моделью:

- абстрагирование для выделения в моделируемом предмете важного для решения конкретной задачи по предмету, в конечном счете - контекстное понимание предмета, формализуемое в виде класса; - инкапсуляция для быстрой и безопасной организации собственно иерархической управляемости: чтобы было достаточно простой команды «что делать», без одновременного уточнения как именно делать, так как это уже другой уровень управления; - наследование для быстрой и безопасной организации родственных понятий: чтобы было достаточно на каждом иерархическом шаге учитывать только изменения, не дублируя все остальное, учтенное на предыдущих шагах; - полиморфизм для определения точки, в которой единое управление лучше распараллелить или наоборот - собрать воедино.

---

[4]Гради Буч. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. 2-е издание, Издательство: Бином, Невский Диалект, 1998, ISBN 0-8053-5340-2, ISBN 5-7989-0067-3, ISBN 5-7940-0017-1

[5]Edsger W. Dijkstra Программирование как вид человеческой деятельности. 1979 (EWD117)

То есть фактически речь идет о прогрессирующей организации информации согласно первичным семантическим критериям: «важное/неважное», «ключевое/подробности», «родительское/дочернее», «единое/множественное». Прогрессирование, в частности, на последнем этапе дает возможность перехода на следующий уровень детализации, что замыкает общий процесс.

Обычный человеческий язык в целом отражает идеологию ООП, начиная с инкапсуляции представления о предмете в виде его имени и заканчивая полиморфизмом использования слова в переносном смысле, что в итоге развивает[6] выражение представления через имя предмета до полноценного понятия-класса.

Процедурное программирование

Процедурное программирование – программирование на императивном языке, при котором последовательно выполняемые операторы можно собрать в подпрограммы, то есть более крупные целостные единицы кода, с помощью механизмов самого языка[7].

Процедурное программирование является отражением архитектуры традиционных ЭВМ, которая была предложена Фон Нейманом в 1940-х годах. Теоретической моделью процедурного программирования служит абстрактная вычислительная система под названием машина Тьюринга.

Автоматное программирование

Автоматное программирование – это парадигма программирования, при использовании которой программа или её фрагмент осмысливается как модель какого-либо формального автомата. Известна также и другая "парадигма автоматного программирования, состоящая в представлении сущностей со сложным поведением в виде автоматизированных объектов управления, каждый из которых представляет собой объект управления и автомат". При этом о программе, как в автоматическом управлении, предлагается думать как о системе автоматизированных объектов управления.

В зависимости от конкретной задачи в автоматном программировании могут использоваться как конечные автоматы, так и автоматы с более сложным строением.

Определяющими для автоматного программирования являются следующие особенности:

временной период выполнения программы разбивается на шаги автомата, каждый из которых представляет собой выполнение определённой (одной и той же для каждого шага) секции кода с единственной точкой входа; такая секция может быть оформлена, например, в виде отдельной функции и может быть разделена на подсекции, соответствующие отдельным состояниям или категориям состояний передача информации между шагами автомата осуществляется

[6] Л.В. Успенский. "Слово о словах". - 5-е изд. - Л.: Детская литература (Ленинградское отделение), 1971

[7] Хювёнен, Сеппянен, 1990, т. 2, с. 27.

только через явно обозначенное множество переменных, называемых состоянием автомата; между шагами автомата программа (или её часть, оформленная в автоматном стиле) не может содержать неявных элементов состояния, таких как значения локальных переменных в стеке, адреса возврата из функций, значение текущего счётчика команд и т. п.; иначе говоря, состояние программы на любые два момента входа в шаг автомата могут различаться между собой только значениями переменных, составляющих состояние автомата (причём такие переменные должны быть явно обозначены в качестве таковых). Полностью выполнение кода в автоматном стиле представляет собой цикл (возможно, неявный) шагов автомата.

Название автоматное программирование оправдывается ещё и тем, что стиль мышления (восприятия процесса исполнения) при программировании в этой технике практически точно воспроизводит стиль мышления при составлении формальных автоматов (таких как машина Тьюринга, автомат Маркова и др.)

Сфера применения: Автоматное программирование широко применяется при построении лексических анализаторов (классические конечные автоматы) и синтаксических анализаторов (автоматы с магазинной памятью)[8].

Кроме того, мышление в терминах конечных автоматов (то есть разбиение исполнения программы на шаги автомата и передача информации от шага к шагу через состояние) необходимо при построении событийно-ориентированных приложений. В этом случае программирование в стиле конечных автоматов оказывается единственной альтернативой порождению множества процессов или потоков управления (тредов).

Часто понятие состояний и машин состояний используется для спецификации программ. Так, при проектировании программного обеспечения с помощью UML для описания поведения объектов используются диаграммы состояний (state machine diagrams). Кроме того, явное выделение состояний используется в описании сетевых протоколов (см., например, RFC 793[9]).

Мышление в терминах автоматов (шагов и состояний) находит применение и при описании семантики некоторых языков программирования. Так, исполнение программы на языке Рефал представляет собой последовательность изменений поля зрения Рефал-машины или, иначе говоря, последовательность шагов Рефал-автомата, состоянием которого является содержимое поля зрения (произвольное Рефал-выражение, не содержащее переменных).

Механизм продолжений языка Scheme для своей реализации также требует мышления в терминах состояний и шагов, несмотря на то что сам язык Scheme никоим образом не является автоматным. Тем не менее, чтобы обеспечить возможность «замораживания» продолжения, приходится при реализации вычислительной модели языка Scheme объединять все компоненты среды исполнения, включая список действий, которые осталось выполнить для окончания вычислений, в единое целое, которое также обычно называется продолжением. Такое продолжение оказывается состоянием автомата, а процесс выполнения программы состоит из шагов, каждый из которых выводит следующее значение продолжения из

[8] А. Ахо, Дж. Ульман. Теория синтаксического анализа, перевода и компиляции = The theory of parsing, translation and compiling. - М.: МИР, 1978. - Т. 1. - 612 с

[9] Postel, J., ed., Transmission Control Protocol, RFC 793

предыдущего.

Александр Оллонгрен в своей книге[3] описывает так называемый Венский метод описания семантики языков программирования, основанный целиком на формальных автоматах.

В качестве одного из примеров применения автоматной парадигмы можно назвать систему STAT[10]; эта система, в частности, включает встроенный язык STATL, имеющий чисто автоматную семантику.

Существуют также предложения по использованию автоматного программирования в качестве универсального подхода к созданию компьютерных программ вне зависимости от предметной области. Так, авторы статьи[11] утверждают, что автоматное программирование способно сыграть роль легендарной серебряной пули

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

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

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

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

[10] А. Ахо, Дж. Ульман. Теория синтаксического анализа, перевода и компиляции = The theory of parsing, translation and compiling. - М.: МИР, 1978. - Т. 1. - 612 с.

[11] Туккель Н.И., Шалыто А.А. Программирование с явным выделением состояний // Мир ПК. - 2001. - № 9. - С. 132-138

Описание объекта управления

Детальное описание объекта, проработка функций и требований как никак лучше может охарактеризовать автоматный подход к разработке, целью которого служит точное понимание системы еще на этапе проектирования.

Функциональные требования

Диаграма состояний

Диаграмма состояний – это, по существу, диаграмма состояний из теории автоматов со стандартизированными условными обозначениями[12] [13], которая может определять множество систем от компьютерных программ до бизнес-процессов. Используются следующие условные обозначения:

- Круг, обозначающий начальное состояние.
- Окружность с маленьким кругом внутри(известная как «кошачий глаз»), обозначающая конечное состояние (если есть).
- Скруглённый прямоугольник, обозначающий состояние. Верхушка прямоугольника содержит название состояния. В середине может быть горизонтальная линия, под которой записываются активности, происходящие в данном состоянии.
- Стрелка, обозначающая переход. Название события (если есть), вызывающего переход, отмечается рядом со стрелкой. Охраняющее выражение может быть добавлено перед "/"и заключено в квадратные скобки (название_события[охраня
что значит, что это выражение должно быть истинным, чтобы переход имел место. Если при переходе производится какое-то действие, то оно добавляется после "/"(название_события[охраняющее_выражение] / действие).
- Толстая горизонтальная линия с либо множеством входящих линий и одной выходящей, либо одной входящей линией и множеством выходящих. Это обозначает объединение и разветвление соответственно.

[12] OMG. OMG Unified Modeling Language (OMG UML), Infrastructure Version 2.2 (February 2009)

[13] D. Drusinsky, Modelling and verification using UML statecharts, Elsevier, 2006

Таблица состояний

Блок-схема

Функциональная схема

При описании сложной системы управления ее разделяют на подсистемы, модули. Нрафическое представление взаимодействия компонент системы пожно представить при помощи функциональной схемы.

Верификация программ методом Model Checking

> Тестирование программы может весьма эффективно продемонстрировать наличие ошибок, но безнадежно неадекватно для демонстрации их отсутствия
>
> Эдсгер Вибе Дейкста

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

Системы переходов [5]

Понятие системы переходов

Системой переходов(СП) называется пятерка S вида

$$S = (P, Q, \delta, L, Q^0) \tag{1}$$

компоненты которой имеют следующий смысл.

1. P - множество, элементы которого называются **утверждениями**
2. Q - множество, элементы которого называются **состояниями СП** S
3. δ - ьинарное отношение на Q(т.е. $\delta \supseteq Q \times Q$) называемое **отношением перехода**
4. L - функция вида
$$L : Q \times P \to \{0, 1\} \tag{2}$$
называемая **оценкой**, которая имеет следующий смысл: для каждого $q \in Q$ и каждого $\rho \in P$ утверждение ρ считается
   - **истинным** в состоянии q, если $L(q, \rho) = 1$,
   - **ложным** в состоянии q, если $L(q, \rho) = 0$

   выражение $L(q, \rho)$ может записываться более компактно в виде знакосочетания $p(\rho)$
5. $Q^0 \in Q$ - множество **начальных состояний**

Темпоральная логика

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

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

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

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

Наиболее известны темпоральные логики

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

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

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

$$\neg\psi, \psi \wedge \eta, \psi \vee \eta \tag{3}$$

тоже принадлежат логике Ф.

Формулы (3) называются булевыми комбинациями формул ψ и η.

LTL

CTL

Язык Promela

`PROMELA (Process or Protocol Meta Language)` – это язык описания описания моделей верификации, созданный `Gerard J. Holzmann` [8]. Язык поддерживает создание процессов для проверки распределенных моделей. Модели в языке могут взаимодействовать между собой при помощи каналов сообщений как в синхронном 2режиме, так и в асинхронном. Модели описанные при помощи языка могут быть обработаны и проанализированы SPIN о чем будет рассказано в последующих главах. Существуют иные реализации и утилиты использующие язык `Promela`, но пока они рассматриваться не будут.

В основном, язык предназначен для проверки логики работы парраллельных систем. Модели описанные `Promela` и обработанные утилитой `SPAN` проверяют модель на корректность в режиме случайной или последовательной симуляции или генерируют код на `C` для быстрой и полной проверки в системном окружении. В процессе симуляции и проверки `SPIN` проверяет отсутствия `deadlocks`[14],

[14] `https://ru.wikipedia.org/wiki/Deadlock`

неопределенных состояний и неиспользуемых частей кода. Также данный подход может проверять правильность системных инвариантов[15], а также поиска зацикливаний и неправильных ветвлений. Также он поддерживает проверку LTL ограничений.

Список спецификаций различных систем, модель которых описана на языке Promela приведена в статье Alberto Lluch[16]

Типы данных

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

Типы bit и bool это синонимы.

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

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

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

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

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

Процессы

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

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

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

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

[15] https://en.wikipedia.org/wiki/Invariant_(computer_science)
[16] http://www.albertolluch.com/research/promelamodels

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

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

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

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

Атомарные конструкции

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

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

Каналы сообщений

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

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

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

```
qname ! expression;
```

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

```
qname ? msg;
```

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

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

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

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

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

Ветвления и конструкции управления

Простейшее сравнение двух переменных:

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

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

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

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

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

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

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

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

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

Циклы

Для повторения группы выражений применяются циклы. Пример

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

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

Безусловные переходы

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

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

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

Проверки

Выжной частью модели описанной языком `Promela` является утверждение

```
assert (any_boolean_condition)
```

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

Составные типы данных

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

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

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

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

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

Исполоняемость

Исполняемость модели обеспечивает базовые средства языка для моделирования синхронизации процессов.

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

proctype  P2 ( chan  Chan_data_in, Chan_data_out) {
     do
    ::  Chan_data_in  ? M_DW ->  skip ;
    ::  Chan_data_out ! M_UP ->  skip ;
     od ;
};

init  {
     atomic  {
         run  P1 (Chan_data_up,   Chan_data_down);
         run  P2 (Chan_data_down, Chan_data_up);
    }
}
```

В примере два процесса `P1` и `P2` имеют недетерминированный выбор 1 входа во 2 выход. Возможны два варианта выбора из которых только один будет выбран. Повторение будет бесконечным. При этом модель не получит `deadlock`[17]

Когда SPIN анализирует модель он проверяет ее при помощи недетерминированного алгоритма и проверит все возможные ее состояния. Когда симулятор SPIN будет визуализировать возможные не проверенные связи, он будет использовать генератор случайных чисел, для проверки недетерминированный состояний. Следовательно симулятор может не показать плохие пути выполнения (хотя таких путей в примере нет). Это иллюстрирует разницу между проверкой и симуляцией. Также можно генерировать исполняемый код из моделей `Promela` с использованием `Refinement`[18]

Ключевые слова

Список ключевых слов используемых в языке

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

[17] `https://ru.wikipedia.org/wiki/Deadlock`

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

```
od
of
pc_value
printf
priority
prototype
provided
run
short
skip
timeout
typedef
unless
unsigned
xr
xs
```

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

SPIN

SPIN (англ. Simple Promela Interpreter)[19] - утилита для верификации корректности распределенных программных моделей. Служит для автоматизированной проверки моделей. Развивается Gerard J. Holzmann и его коллегами из Unix group центра Computing Sciences Research Center в Bell Labs начиная с 1980 года. С 1991 года программа распространяется бесплатно вместе с исходными кодами.

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

Описание опций можно посмотреть в приложении Описание опций SPIN

[19] https://en.wikipedia.org/wiki/SPIN_model_checker

Автоматное программирование

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

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

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

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

Тестирование

> Если отладка - процесс удаления ошибок, то программирование должно быть процессом их внесения
>
> Эдсгер Вибе Дейкста

Модульное тестирование[20]

Модульное тестирование, или юнит-тестирование (англ. `unit testing`)[21] - процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

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

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

Существует множество библиотек способствующих к быстрому написанию тестов для языка Си. Мы будем использовать `GoogleTest`[22].

Ключевые понятия

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

[20] В подготовке данной части материала использовалась статья `Google testing framework (gtest)https://habrahabr.ru/post/119090/`

[21] `https://en.wikipedia.org/wiki/Unit_testing`

[22] `https://github.com/google/googletest.git`

Утверждения (assertion)

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

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

- Простейшие логические

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

- Сравнение

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

- Сравнение строк

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

- Проверка на исключения

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

- Проверка предикатов

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

- Сравнение чисел с плавающей точкой

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

- Вызов отказа или успеха

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

Запуск тестов

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

Фалаги

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

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

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

Интеграционное тестирование

Эмуляция устройств

Описание проекта

CMake

CMake (от англ. cross platform make)[23] - это кроссплатформенная система автоматизации сборки программного обеспечения из исходного кода. CMake не занимается непосредственно сборкой, а лишь генерирует файлы управления сборкой из файлов CMakeLists.txt:

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

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

Makefile

[23] https://en.wikipedia.org/wiki/CMake

Практические работы

Каждая практическая работа расчитана на 1-3 практических занятия и соотвествует лабораторным работам.

Описание объекта управления

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

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

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

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

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

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

1. $t < T$ - температура окружающей среды меньше поддерживаемой температуры, поэтому следует перейти в состояние "Нагрев"

2. $t > T$ - температура окружающей среды больше поддерживаемой температуры, поэтому следует перейти в состояние "Охлаждение"

3. $t == T$ - температура окружающей среды соответствует поддерживаемой температуры, поэтому следует перейти в состояние "Ожидание"

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

1. $w > W_1$ - внутренний счетчик времени работы в состоянии "Нагрев"больше предельного значения

2. $w > W_2$ - внутренний счетчик времени работы в состоянии "Охлаждение"больше предельного значения

3. $w > W_3$ - внутренний счетчик времени работы в состоянии "Ожидание"больше предельного значения

при этом, при вхождении в состояние внутренние счетчики сбрасываются.

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

При помощи `PlantUML`[24] постоим диаграмму состояний.

Для описания диаграммы перейдем на сайт `PlantUMLhttp://www.plantuml.com/plantuml/uml` и наберем следующую последовательность символов(если не хочеться вводить, можно перейти по ссылке):

```
@startuml
title  "Работа кондиционера"

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

[24] `https://en.wikipedia.org/wiki/PlantUML`

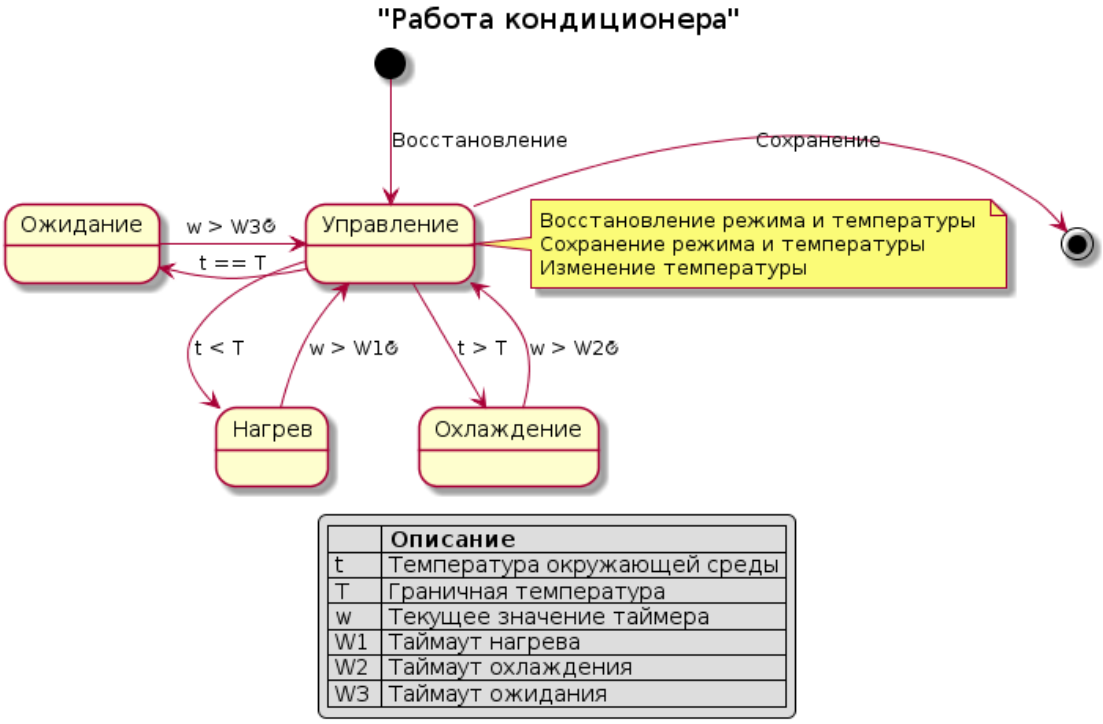

Рис. 1: Диаграмам состояний работы кондиционера

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

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

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

После нажатия на клаыишу "Submit"мы получим следующую диаграму:

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

`right of` добавляет подсказку к состоянию.

Подробности использования языка `PlantUML` можно найти по ссылке `http://plantuml.com/PlantUML_Language_Reference_Guide.pdf`

Модель Promela

Описанный выше объект управления формализуем при помощи Язык Promela.

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

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

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

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


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

Определено 7 состояний:

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

Выделим в нашей моделе отдельный процесс для получения температуры

```
mtype  = {START, ON, OFF, CONTROL, COOLING, HEATING, WAITING,
                  T_READ_PROPERTIES, T_READ, T_WAIT, T_OFF};
```

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

ltl  temp_equals   {
            []<> (current_temperature == guard_temperature) }
ltl  cooling       {
            <> (state == COOLING && current_temperature > guard_temperature) }
ltl  heating       {
            <> (state == HEATING && current_temperature < guard_temperature) }
ltl  off_complete  {
            []<> ( (last_state == WAITING) && (state == OFF) ) }

int   t_wait_count = 2;

proctype  termometer()
  printf("[Termometer %d] Enter\n", _pid);
   do
    :: (t_state == T_READ_PROPERTIES) ->
      printf("[Termometer %d] Load properties\n", _pid);
      t_wait_count = 2;
      t_state = T_READ;
    :: (t_state == T_OFF) ->
      printf("[Termometer %d] Signal off\n", _pid);
       break ;
    :: (t_state == T_READ) ->
      printf("[Termometer %d] Read current temperature: %d\n",
             _pid, current_temperature);
      t_state = T_WAIT;
    :: (t_state == T_WAIT) ->
      t_wait_count--;
       if
        :: (t_wait_count <= 0) ->
          t_wait_count = 2;
          t_state = T_READ;
        ::  else  ->
          t_state = T_WAIT;
       fi
    :: (state == OFF) ->
      t_state = T_OFF;
   od
  printf("[Termometer %d] Exit\n", _pid);
```

```
proctype  conditioner()
  printf("[Conditioner %d] Enter\n", _pid);
   run  termometer();
   do
    :: (state == OFF) ->
      printf("[Conditioner %d] Store temperature and properties\n", _pid);
      t_state == T_OFF;
      printf("[Conditioner %d] Dependencies processes, wait\n", _pid);
       break ;
    :: (state == ON) ->
      current_temperature = 14;
      t_state = T_READ_PROPERTIES;
      printf("[Conditioner %d] Load properties\n", _pid);
      guard_temperature = 12;
      printf("[Conditioner %d] Loaded: %dC, Current: %dC\n",
             _pid, guard_temperature, current_temperature);
      last_state = state;
      state = CONTROL;
    :: (state == CONTROL) ->
      printf("[Conditioner %d] Controling\n", _pid);
      printf("[Conditioner %d] Current temperature: %dC\n",
             _pid, current_temperature);
      last_state = state;
       if
        :: (current_temperature > guard_temperature) ->
          state = COOLING;
        :: (current_temperature < guard_temperature) ->
          state = HEATING;
        ::  else  ->
          state = WAITING;
       fi
    :: (state == COOLING) ->
      assert(current_temperature > guard_temperature);
      printf("[Conditioner %d] Cooling\n", _pid);
      current_temperature--;
      last_state = state;
      state = CONTROL;
    :: (state == HEATING) ->
      assert(current_temperature < guard_temperature);
      printf("[Conditioner %d] Heating\n", _pid);
      current_temperature++;
      last_state = state;
      state = CONTROL;
    :: (state == WAITING) ->
      assert(current_temperature == guard_temperature);
```

```
      printf("[Conditioner %d] Waiting\n", _pid);
      last_state = state;
      state = CONTROL;
       if
        :: (waiting_count == 5) ->
          printf("[Conditioner %d] Simulating. Downgrade temperature.\n", _pid);
          current_temperature = guard_temperature - 2;
          waiting_count--;
        :: (waiting_count < 0) ->
          state = OFF;
        ::  else  ->
          waiting_count--;
       fi
   od
  printf("[Conditioner %d] Exit\n", _pid);



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

Проект Си

Листинг 1: Пример управления состояниями на C

```
#include "conditioner.h"

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

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

Модульное тестирование

Обработка ошибок

Логирование

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

Эмулирование внешних устройств

Лабораторные работы

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

Задания

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

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

Требования к оформлению кода

Отступы

* Для обозначения отступа используйте 4 пробела подряд;
* Используйте проблелы, а не табуляцию.

Объявление переменных

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

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

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

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

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

Пробелы

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

* Всегда используйте один пробел перед фигурной скобкой:

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

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

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

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

* Бинарные операции отделяются пробелами с 2-х строн;

* После преобразования типов не ставьте пробелов;

* Избегайте проеобразования типов в стиле C, если ваш код на C++:

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

Фигурные скобки

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

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

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

* Исключение: Тело функции и описание класса всегда открывается фигурной скобкой, стоящей на тойже строке:

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

class Moo {
};
```

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

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

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

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

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

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

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

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

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

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

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

```
    ...

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

* Используйте фигурные скобки для обозначения пустого тела условия:

```
// Wrong
while (a);

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

Круглые скобки

* Используйте круглые скобки для группировки выражений:

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

// Correct
if ((a && b) || c)

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

Использование конструкции `switch`

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

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

```
}
```

Разрыв строк

* Длина строки кода не должна превышать 100 символов. Если надо – используйте разрыв строки.

* Запятые помещаются в конец разорванной линии; операторы помещаются в начало новой строки. В зависимости от используемой вами `IDE`, оператор на конце разорванной строки можно проглядеть:

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

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

Наследование и ключево слово `virtual`

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

Главное исключение

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

Требования к оформлению работы

Лабораторная работа должна состоять из следующих необходимых компонент:

1 лист Титульный лист

2 лист Описание объекта управления(цели и задачи объекта управления, функциональные требования, граничные условия, параметры и их назаначение). Пример Описание объекта управления.

3 лист Модель `Promela`. Пример Модель `Promela`.

4 лист Диаграмма переходов. Пример Описание объекта управления.

5 лист Описание модулей, компонент их взаимодействие.

Работа считается принятой если выполнены условия:

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

Приложение

Б.П. Кузнецов об автоматном программировании

Б. П. Кузнецов[25]

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

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

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

[25]к.т.н Концерн «НПО «Аврора», г. Санкт-Петербург

12. Не учтенные выходные воздействия, в особенности как реакция на ошибочное поведение управляемого объекта и самого управляющего автомата и связи управляющего и операционного автоматов с объектом управления;

13. Неверная пометка состояний и переходов буквами выходного алфавита;

14. Неверное отождествление букв выходного алфавита с выходными воздействиями;

15. Забытое обнуление или продление выходного воздействия;

16. Путаница в отметке переходов и состояний буквами выходного алфавита при использовании совмещенной модели (Мили и Мура) автоматов;

17. Не прослеживаются полные пути в диаграмме состояний;

18. Всевозможные ошибки в полных путях при использовании обратных связей в диаграмме состояний

19. Другие ошибки.

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

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

Описание опций SPIN

Run Time Options for PAN

-A	suppress the reporting of assertion violations (see also **-E**)
-a	find acceptance cycles (available if compiled *without* **-DNP**)
-b	bounded search mode, makes it an error to exceed the search depth, triggering and error trail
-cN	stop at *N* th error (defaults to first error if *N* is absent)
-d	print state tables and stop
-E	suppress the reporting of invalid endstate errors (see also **-A**)
-e	create trails for all errors encountered (default is first one only)
-f	add weak fairness (to **-a** or **-l**)
-hN	choose another hash-function, with *N*: 1..32 (defaults to 1)
-i	search for shortest path to error (causes an increase of complexity)
-l	like **-i**, but approximate and faster
-J	reverse the evaluation order of nested unless statements (to conform to the one used in Java)
-kN	set the number of hashfunctions used in bitstate hashing mode to *N* (requires compilation with **-DBITSTATE**) The default is k=2. This option was introduced in version 4.2.0.
-l	find non-progress cycles (requires compilation with **-DNP**)
-mN	set max search depth to *N* steps (default *N*=10000)
-n	no listing of unreached states at the end of the run
-q	require empty channels in valid endstates
-r, -P, -C	play back error trail with embedded C code statements
-s	use 1-bit hashing (default is 2-bit hashing, assumes compilation **-DBITSTATE**). In version 4.2.0 and later, the option -s is equivalent to -k1.
-V	print *Spin* version number and stop
-wN	use ahashtable of 2^*N* entries(defaults to **-w***18*)

Compile Time Options for PAN

Directives Supported by Xspin

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

Directives Related to Partial Order Reduction

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

Directives to Increase Speed

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

Directives to Reduce Memory Use

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

Directives Reserved for Use When Prompted by PAN

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

Directives for Debugging PAN Verifiers

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

Directives for Experimental Use

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

Compile Time Options for SPIN

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

Run Time Options for SPIN

Simulation

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

Verification Generation

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

Source: http://spinroot.com/spin/Man/index.html *Prepared by A. Burak Gurdag on 15.03.2007*

BNF Языка Promela

```
spec : module [ module ] *

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

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

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

never : NEVER '{' sequence '}'

trace : TRACE '{' sequence '}'

utype : TYPEDEF name '{' decl_lst '}'

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

decl_lst: one_decl [ ';' one_decl ] *

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

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

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

priority: PRIORITY const /* simulation priority */

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

visible : HIDDEN | SHOW

sequence: step [ ';' step ] *

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

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

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

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

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

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

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

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

arg_lst  : any_expr [ ',' any_expr ] *

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

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

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

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

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

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

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

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

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

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

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

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

chanpoll: FULL | EMPTY | NFULL | NEMPTY

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

```
uname : name

name : alpha [ alpha | number ] *

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

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

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

Примеры исходного кода

Листинг 2: Пример C

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

Литература

[1] В. Э. Карпов. Автоматное программирование и робототехника.

[2] А. А. Шалыто and Ю. Ю. Янкин. Автоматное программирование плис в задачах управления эл

[3] А. А. Шалыто and С. Э. Вельдер. О ВЕРИФИКАЦИИ ПРОСТЫХ АВТОМАТНЫХ ПРОГРАМ
УДК 681.3.06.

[4] А. А. Шалыто and Н. И. Туккель. Switch-ТЕХНОЛОГИЯ — АВТОМАТНЫЙ ПОДХОД К СОЗ
2000.

[5] А. М. Миронов. Верификация методом model checking.

[6] И. А. Лагунов. ТЕКСТОВЫЙ ЯЗЫК АВТОМАТНОГО ПРОГРАММИРОВАНИЯ fsml ДЛЯ И
2000. УДК 004.432.4.

[7] А. А. Шалыто. АВТОМАТНОЕ ПРОГРАММИРОВАНИЕ, 2006.
При поддержке Министерства образования и науки Российской Федерации в рамках научно-исс

[8] Unknown. Promela. `https://en.wikipedia.org/wiki/Promela`.

Список иллюстраций